**Comando de java script**

**Usar //** le dir a JavaScript que ignore el resto del texto en la lnea actual. Este es un comentario en línea:

**// This is an in-line comment.**

**Puedes hacer un comentario multilínea comenzando** con /\*y terminando con \*/. **Este es un comentario multilínea:**

/\* This is a

multi-line comment \*/

**Cuando una variable ha quedado sin definir tiene un valor especial denominado undefined** , que se podría traducir como “sin definir“. Este valor puede resultar un poco desconcertante, ya que además de un tipo de datos, existe una variable con ese nombre y también es un valor como tal.

**El valor null es un literal de Javascript** que representa intencionalmente un valor nulo o "vacío". Es uno de los valores primitivos de Javascript

**Boolean** son un tipo de dato primitivo comúnmente usado en lenguajes de programación. Por definición, un booleano tiene 2 posibles valores: true (verdadero) o false (falso).

**El tipo String de JavaScript** se utiliza para representar datos textuales. Es un conjunto de "elementos" de valores enteros sin signo de 16 bits (unidades de código UTF-16). Cada elemento de la cadena de caracteres ocupa una posición en la cadena.

**Symbol value o simplemente Symbol** — que está garantizado que sea único. Los Symbols se utilizan a menudo para agregar claves de propiedades únicas a un objeto que no son iguales a las claves que cualquier otro código puede agregar al objeto, y que están ocultas de cualquier mecanismo que otro código utilice normalmente para acceder al objeto.

**BigInt** es un tipo de dato numerico que puede representar números enteros en el formato de precision arbitrario. En otros lenguajes de programación pueden existir diferentes tipos numéricos, por ejemplo: enteros, flotantes, dobles o bignums (numeros grandes).

JavaScript, number se implementan en Formato en coma flotante de doble precisión de 64 bits IEEE 754 (es decir, un número entre ±2−1022 y ±2+1023, o aproximadamente ±10−308 a ±10+308, con una precisión numérica de 53 bits). Los valores enteros hasta ±253 - 1 se pueden representar con exactitud.

En JavaScript, **las funciones son object** de primera clase, es decir, son objetos y se pueden manipular y transmitir al igual que cualquier otro objeto. Concretamente son objetos Function

Una palabra clave llamada **let** fue abandonada 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.

La palabra clave const **especifica que el valor de una variable es constante e indica al compilador que evite que el programador lo modifique**. En C, los valores constantes tienen la vinculación externa como valor predeterminado, por lo que solo pueden aparecer en los archivos de código fuente

**Suma dos numeros 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 acceso cuando se coloca entre dos números.

**const sum = 10 + 10; función para sumar**

**Resta un número de otro con JavaScript**

También podemos restaurar un número de otro.

JavaScript utiliza el símbolo -para reiniciar.

**const difference = 45 - 33;** **para restar**

**Multiplica dos números con JavaScript**

También podemos multiplicar un número por otro.

JavaScript utiliza el símbolo \*para multiplicar dos números.

**const product = 8 \* 10;**

**Divide un número entre otro con JavaScript**

También podemos dividir un número entre otro.

JavaScript utiliza el símbolo /para la división.

**const quotient = 66 / 33;**

**Incrementa un número con JavaScript**

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

**myVar++;**

**Decrementa un número con JavaScript**

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

**myVar--;**

**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: cuando calcula números, se calcula con precisión finita. Los puntos con operaciones flotantes pueden producir resultados diferentes a los deseados.

**var myDecimal = 2.8;**

**Multiplica dos números decimales con JavaScript**

En JavaScript, también puedes realizar cálculos con números decimales, al igual que con números enteros.

Multiplicamos dos números decimales para obtener su producto.

**const product = 1.0 \* 5.0;**

**Divide un decimal entre otro con JavaScript**

Ahora dividimos un decimal entre otro.

**const quotient = 4.4 / 2.0;**

**Encuentra un resto en JavaScript**

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

**ejemplo**

5 % 2 = 1

5 / 2 = 2 resto 1

2 \* 2 = 4

5 - 4 = 1

**Uso**

En matemáticas, se puede comprobar si un número es par o impar comprobando el resto de la división del número entre 2. Los números pares tienen un resto de 0, mientras que los números impares tienen un resto de 1.

17 % 2 = 1

48 % 2 = 0

**Nota:** El operador de resto es a veces incorrectamente referido como el módulo operador. Es muy similar al módulo, pero no funciona adecuadamente con números negativos.

**var remainder = 11 % 3; //remainder gets the value 2.**

**Asignación compuesta con aumentada 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 5a myVar. Dado que se trata de un patrón tan común, hay operadores que hacen tanto la operación matemática como la modificada en un solo paso.

**Uno de estos operadores es el operador +=.**

let myVar = 1;

myVar += 5;

console.log(myVar);

**let a = 3;**

**let b = 17;**

**let c = 12;**

**// Cambia solo el código debajo de esta línea**

**a += 12;**

**b += 9;**

**c += 7;**

**Asignación compuesta con resto aumentado**

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

**myVar = myVar - 5;**

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

**myVar -= 5;**

**let a = 11;**

**let b = 9;**

**let c = 3;**

**// Cambia solo el código debajo de esta línea**

**a -= 6;**

**b -= 15;**

**c -= 1;**

**Asignación compuesta con multiplicación aumentada**

El operador \*=multiplica una variable por un número.

**myVar = myVar \* 5;**

se multiplicará myVarpor 5. Esto se puede reescribir como:

**myVar \*= 5;**

**let a = 5;**

**let b = 12;**

**let c = 4.6;**

**// Cambia solo el código debajo de esta línea**

**a \*= 5;**

**b \*= 3;**

**c \*= 10;**

**Asignación compuesta con división aumentada**

El operador /=divide una variable entre otro número.

**myVar = myVar / 5;**

Divide myVarpor 5. Esto se puede reescribir como:

**myVar /= 5;**

**let a = 48;**

**let b = 108;**

**let c = 33;**

**// Cambia solo el código debajo de esta línea**

**a /= 12;**

**b /= 4;**

**c /= 11;**

**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 requerir 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".**

**const myStr = "I am a \"double quoted\" string inside \"double quotes\"."; // Cambia esta línea**

**citar cadenas con comillas simples**

Los valores de cadena en JavaScript se pueden escribir con comillas simples o dobles, siempre que comience y termine 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 podría querer usar un tipo de comillas sobre la otra es si desea usar ambas en una cadena. Esto podría suceder si desea guardar una conversación en una cadena y tener la conversación entre comillas. Otro uso sería guardar una <a>etiqueta 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 si necesita usar las comillas más externas dentro de él. Recuerde, una cadena tiene el mismo tipo de comillas al principio y al final. Pero si tiene la misma cita en algún lugar en el medio, la cadena se detendrá antes de tiempo y generará un error.

**const goodStr = 'Jake asks Finn, "Hey, let\'s go on an adventure?"';**

**const badStr = 'Finn responds, "Let's go!"';**

**var myStr = '<a href="http://www.example.com" target="\_blank">Link</a>';**

**Secuencias de escape en cadenas**

Las comillas no son los únicos caracteres que se pueden escapar dentro de una cadena. Las secuencias de escape le permiten usar caracteres que de otro modo no podría usar en una cadena.

**Código Producción**

**\' una frase**

**\" comillas dobles**

**\\ barra invertida**

**\n nueva línea**

**\t pestaña**

**\r retorno de carro**

**\b límite de palabras**

**\f alimentación de formulario**

Tenga en cuenta que la barra invertida en sí debe tener escape para que se muestre como una barra invertida.

**Concatenación de cadenas con el operador Plus**

En JavaScript, cuando el +operador se usa con un Stringvalor, se llama operador de concatenación . Puede crear una nueva cadena a partir de otras cadenas concatenándolas .

**Ejemplo**

**'My name is Alan,' + ' I concatenate.'**

**Nota:** Cuidado con los espacios. La concatenación no agrega espacios entre cadenas concatenadas, por lo que deberá agregarlos usted mismo.

**Ejemplo:**

**const ourStr = "I come first. " + "I come second.";**

**// Example**

**var ourStr = "I come first. " + "I come second.";**

**// Only change code below this line**

**const myStr = "This is the start. " + "This is the end."**

**console.log(myStr);**

**concatenación de cadenas con el operador más igual a**

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

Nota: Cuidado con los espacios. La concatenación no agrega espacios entre cadenas concatenadas, por lo que deberá agregarlos usted mismo.

**Ejemplo:**

**let ourStr = "I come first. ";**

**ourStr += "I come second.";**

**var myStr = "This is the first sentence. "**

**myStr += "This is the second sentence."**

**Construcción de cadenas con variables**

A veces necesitará construir una cadena. Al usar el operador de concatenación ( +), puede insertar una o más variables en una cadena que está creando.

**Ejemplo:**

**const ourName = "freeCodeCamp";**

**const ourStr = "Hello, our name is " + ourName + ", how are you?";**

**const myName = "ulises";**

**const myStr = "My name is " + myName + " and I am well!";**

**Agregar variables a cadenas**

Así como podemos construir una cadena en varias líneas a partir de literales de cadena , también podemos agregar variables a una cadena usando el +=operador más igual ( ).

**Ejemplo:**

**const anAdjective = "awesome!";**

**let ourStr = "freeCodeCamp is ";**

**ourStr += anAdjective;**

**const someAdjective = "ulises";**

**let myStr = "Learning to code is ";**

**myStr += someAdjective;**

**Encuentra la longitud de una cadena**

Puede encontrar la longitud de un Stringvalor escribiendo .lengthdespués de la variable de cadena o el literal de cadena.

**console.log("Alan Peter".length);**

**// Setup**

**let lastNameLength = 0;**

**const lastName = "Lovelace";**

**// Only change code below this line**

**lastNameLength = lastName.length;**

**console.log(lastNameLength)**

**Use 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 específico dentro de una cadena.

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

Por ejemplo, el carácter en el índice 0 de la palabra Charleses C. Entonces const firstName = "Charles", si puede obtener el valor de la primera letra de la cadena usando firstName[0].

**Ejemplo:**

**const firstName = "Charles";**

**const firstLetter = firstName[0];**

**// Only change code below this line**

**firstLetterOfLastName = lastName[0]; // Change this line**

**console.log(firstLetterOfLastName)**

**Comprender la inmutabilidad de cadenas**

En JavaScript, Stringlos valores son inmutables , lo que significa que no se pueden modificar una vez creados.

**Por ejemplo, el siguiente código generará un error porque la letra Bde la cadena Bobno se puede cambiar a la letra J:**

let myStr = "Bob";

myStr[0] = "J";

**// Setup**

**let myStr = "Jello World";**

**// Only change code below this line**

**myStr = "Hello World"; // Change this line**

**// Only change code above this line**

**Utiliza la notación de corchetes para encontrar el enésimo personaje 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 personaje es en realidad el personaje cero.

**Ejemplo:**

**const firstName = "Ada";**

**const secondLetterOfFirstName = firstName[1];**

**// Configuración**

**const lastName = "Lovelace";**

**// Cambia solo el código debajo de esta línea**

**const thirdLetterOfLastName = lastName[2]; // Cambia esta línea**

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

Con el fin de obtener la última letra de una cadena, puedes restaurar 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];**

**// Configuración**

**const lastName = "Lovelace";**

**// Cambia solo el código debajo de esta línea**

**const lastLetterOfLastName = lastName[lastName.length - 1]; // Cambia esta línea**

**Utiliza la notación de corchetes para encontrar el personaje 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"usandofirstName[firstName.length - 3]

**Ejemplo:**

**const firstName = "Augusta";**

**const thirdToLastLetter = firstName[firstName.length - 3];**

**// Configuración**

**const lastName = "Lovelace";**

**// Cambia solo el código debajo de esta línea**

**const secondToLastLetterOfLastName = lastName[lastName.length - 2]; // Cambia esta línea**

**Palabra en blanco**

Se te garantizan oraciones con algunas palabras faltantes, como sustantivos, verbos, adjetivos y adverbios. Luego, completa las piezas que faltan con palabras de tu elección de una manera que la oración completa tenga sentido.

**Considere esta frase:**

It was really \_\_\_\_, and we \_\_\_\_ ourselves \_\_\_\_.

A esta frase le faltan tres piezas: un adjetivo, un verbo y un adverbio, y podemos añadir las palabras que queramos para completarla. A continuación, podemos asignar la frase completa a una variable de la siguiente manera:

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

**const myNoun = "dog";**

**const myAdjective = "big";**

**const myVerb = "ran";**

**const myAdverb = "quickly";**

**// Cambia solo el código debajo de esta línea**

**const wordBlanks = "The " + myAdjective + " " + myNoun + " " + myVerb + " " + myAdverb + ".";; // Cambia esta línea**

**// Cambia solo el código encima de esta línea**

**console.log(wordBlanks);**

**Almacena múltiples valores en una variable usando 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"];**

**const myArray = ["ulises",23];**

**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.

**const myArray = [["ulises",23], ["dili",22]];**

**acceda 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 comenzando desde cero , en donde el primer elemento de un arreglo tiene como índice 0.

**ejemplo**

**const array = [50, 60, 70];**

**console.log(array[0]);**

**const data = array[1];**

**console.log(array[0])imprime 50, y datatiene el valor 60..**

**var ourArray = [50,60,70];**

**var ourData = ourArray[0]; // equals 50**

**// Setup**

**var myArray = [50,60,70];**

**// Only change code below this line.**

**var myData = myArray[0];**

**const myArray = [45, 64, 99];**

**myArray[0] = 45;**

**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 un arreglo, el primer par de corchetes hace referencia a los elementos del arreglo más externo (el primer nivel), y cada par adicional va haciendo referencia a un nivel más interno.

**ejemplo**

**const arr = [**

**[1, 2, 3],**

**[4, 5, 6],**

**[7, 8, 9],**

**[[10, 11, 12], 13, 14]**

**];**

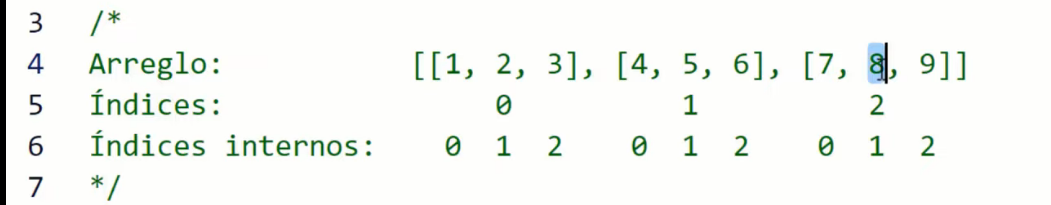
**const subarray = arr[3];**

**const nestedSubarray = arr[3][0];**

**const element = arr[3][0][1];**

En este ejemplo, subarraytiene el valor [[10, 11, 12], 13, 14], nestedSubarraytiene el valor [10, 11, 12], y elementtiene el valor 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, puede confundir a otros programadores al leer su código.**

****

**const myArray = [**

**[1, 2, 3],**

**[4, 5, 6],**

**[7, 8, 9],**

**[[10, 11, 12], 13, 14],**

**];**

**const myData = myArray[2][1];**

**Manipular arreglos con el método push**

Una forma fácil de agregar 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"]);**

**arr1ahora tiene el valor [1, 2, 3, 4]y arr2tiene el valor ["Stimpson", "J", "cat", ["happy", "joy"]].**

**// Configuración**

**const myArray = [["John", 23], ["cat", 2]];**

**// Cambia solo el código debajo de esta línea**

**myArray.push(["dog", 3]);**

**Manipular arreglos con el metodo 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 asignandolo 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.logmuestra el valor 6y el segundo muestra el valor [1, 4].**

**// Setup**

**var myArray = [["John", 23], ["cat", 2]];**

**// Only change code below this line.**

**var removedFromMyArray = myArray.pop();**

**console.log(myArray)**

**Manipular arreglos con el método shift**

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

Ahi 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();**

**removedFromOurArraytendrá una cadena con valor Stimpsony ourArraytendrá ["J", ["cat"]].**

**// Configuración**

**var myArray = [["John", 23], ["dog", 3]];**

**// Only change code below this line.**

**var removedFromMyArray = myArray.shift();**

**Manipular arreglos con el método 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 agregar el elemento al final del arreglo, unshift()agrega el elemento al principio del arreglo.

**Ejemplo:**

**const ourArray = ["Stimpson", "J", "cat"];**

**ourArray.shift();**

**ourArray.unshift("Happy");**

**Después del shift, ourArraytendrá el valor ["J", "cat"]. Después del unshift, ourArraytendrá el valor ["Happy", "J", "cat"].**

**const myArray = [["John", 23], ["dog", 3]];**

**myArray.shift();**

**myArray.unshift(["Paul", 35]);**

**lista de compras**

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

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]**

**const myList = [["carne", 1], ["pasta",2], ["tomate", 4], ["jugos", 4], ["queso", 3]];**

**Escribe JavaScript reutilizable usando 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");**

**}**

Puede 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 Worlden la consola de desarrollo. Todo el código entre las llaves se ejecutará cada vez que se llame la función.

**function reusableFunction() {**

**console.log("Hi World");**

**}**

**reusableFunction();**

**pasa valores a las funciones usando 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 normalmente 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, param1y param2:**

**function testFun(param1, param2) {**

**console.log(param1, param2);**

}

Entonces podemos llamar a testFunasí: testFun("Hello", "World");. Hemos pasado dos argumentos de cadena, Helloy World. Dentro de la función, param1será igual a la cadena Helloy param2será igual a la cadena World. Ten en cuenta que podrías llamar a testFunotra vez con diferentes argumentos y los parámetros tomarían el valor de los nuevos argumentos.

**function functionWithArgs(a, b) {**

**console.log(a + b);**

**}**

**functionWithArgs(1,2)**

**functionWithArgs(7,9)**

**Devuelve un valor de una función usando "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);**

answertiene el valor 8.

plusThreetoma un argumento para numy devuelve un valor igual a num + 3.

**function timesFive(num) {**

**return num \* 5;**

**}**

**console.log(timesFive(5));**

**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 constse** 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.**

**// Declara la variable myGlobal debajo de esta línea**

**let myGlobal = 10;**

**function fun1() {**

**// Assign 5 to oopsGlobal here**

**oopsGlobal = 5;**

**}**

**// Cambia solo el código encima de esta línea**

**function fun2() {**

**let output = "";**

**if (typeof myGlobal != "undefined") {**

**output += "myGlobal: " + myGlobal;**

**}**

**if (typeof oopsGlobal != "undefined") {**

**output += " oopsGlobal: " + oopsGlobal;**

**}**

**console.log(output);**

**}**

**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 myTestcon una llamada local variable loc.**

**function myTest() {**

**const loc = "foo";**

**console.log(loc);**

**}**

**myTest();**

**console.log(loc);**

La llamada a la función myTest()muestra la cadena fooen la consola. La línea console.log(loc)(fuera de la función myTest) lanzará un error, ya que locno está definido fuera de la función.

**function myLocalScope() {**

**// Cambia solo el código debajo de esta línea**

**var myVar = 5;**

**console.log('inside myLocalScope', myVar);**

**}**

**myLocalScope();**

**// Ejecuta y verifica la consola**

**// myVar no está definida afuera de myLocalScope**

**console.log('outside myLocalScope');**

**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 myFundevolverá la cadena Headporque está presente la versión local de la variable.

**var outerWear = "T-Shirt";**

**function myOutfit() {**

**var outerWear = "sweater";**

**return outerWear;**

**}**

**console.log(myOutfit());**

**console.log(outerWear);**

**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);**

addSumes una función sin una declaración de devolución ( return). La función cambiará la variable global sumpero el valor devuelto de la función es undefined.

**// Configuración**

**let sum = 0;**

**function addThree() {**

**sum = sum + 3;**

**}**

**// Cambia solo el código debajo de esta línea**

**function addFive() {**

**sum = sum + 5;**

**}**

**// Cambia solo el código encima de esta línea**

**addThree();**

**addFive();**

**Asignación con un valor devuelto**

Si recuerda nuestra discusión sobre el almacenamiento de valores con el operador de borrado , todo lo que está 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 definido una función sumque suma dos números.

**ourSum = sum(5, 12);**

Llamar a la función sumcon los argumentos 5y 12producir un valor de retorno de 17. Este valor de retorno se asigna a la variable ourSum.

**// Configuración**

**let processed = 2;**

**function processArg(num) {**

**return (num + 3) / 5;**

**}**

**// Cambia solo el código debajo de esta línea**

**processed = processArg(7);**

**Permanece en linea**

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

**function nextInLine(arr, item) {**

**// Cambia solo el código debajo de esta línea**

**arr.push(item);**

**return arr.shift();**

**// Cambia solo el código encima de esta línea**

**}**

**// Configuración**

**let testArr = [1, 2, 3, 4, 5];**

**// Muestra el código**

**console.log("Before: " + JSON.stringify(testArr));**

**console.log(nextInLine(testArr, 6));**

**console.log("After: " + JSON.stringify(testArr));**

**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 trueestá encendido y falseestá apagado. Estos dos estados se excluyen condiciones.

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.

**function welcomeToBooleans() {**

**// Cambia solo el código debajo de esta línea**

**return true; // Cambia esta línea**

**// Cambia solo el código encima de esta línea**

**}**

**Usa condicional logico con las sentencias "If"**

Las sentencias ifson usadas para tomar decisiones en el código. La palabra clave ifle dice a JavaScript que ejecuta el código entre llaves bajo ciertas condiciones, definidas en los paréntesis. Estas condiciones son conocidas como condiciones booleanas ( Boolean) y sólo pueden ser trueo 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 truey test(false)devuelve la cadena It was false.**

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

**// Example**

**function ourTrueOrFalse(isItTrue) {**

**if (isItTrue) {**

**return "Yes, it's true";**

**}**

**return "No, it's false";**

**}**

**function trueOrFalse(wasThatTrue) {**

**if (wasThatTrue) {**

**return "Yes, that was true";**

**}**

**return "No, that was false";**

**}**

**console.log(trueOrFalse(true));**

**Comparación con el operador de igualdad**

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

El operador más básico es el de igualdad ==. El operador de igualdad compara dos valores y devuelve truesi son equivalentes o falsesi no lo son. Ten en cuenta que una igualdad es diferente a una pantalla ( =), 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 myVales 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, numbersy 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 // true**

**1 == 2 // false**

**1 == '1' // true**

**"3" == 3 // true**

**function testEqual(val) {**

**if (val == 12) { // Change this line**

**return "Equal";**

**}**

**return "Not Equal";**

**}**

**// Change this value to test**

**console.log(testEqual(10));+**

**Comparación con el Operador de Igualdad Estricta**

La igualdad estricta ( ===) es la contrapartida del operador de igualdad ( ==). Sin embargo, a diferencia del operador de igualdad, que intenta convertir ambos valores que se comparan con un tipo común, el operador de igualdad estricta no realiza una conversión de tipo.

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

**Ejemplos**

**3 === 3 // true**

**3 === '3' // false**

**En el segundo ejemplo, 3es un Numbertipo y '3'es un Stringtipo.**

**// Setup**

**function testStrict(val) {**

**if (val === 7) { // Change this line**

**return "Equal";**

**}**

**return "Not Equal";**

**}**

**testStrict(10);**

**practica comparando diferentes valores**

En los últimos dos desafíos, aprendimos sobre el operador de igualdad ( ==) y el operador de igualdad estricta ( ===). Hagamos una revisión rápida y practiquemos un poco más el uso de estos operadores.

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 igualdad estricta comparará tanto el tipo de datos como el valor tal cual, sin convertir un tipo en otro.

Ejemplos

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

Nota: en JavaScript, puede determinar el tipo de una variable o un valor con el **typeofoperador, de la siguiente manera:**

**typeof 3**

**typeof '3'**

**typeof 3devuelve la cadena numbery typeof '3'devuelve la cadena string.**

**// Setup**

**function compareEquality(a, b) {**

**if (a === b) { // Change this line**

**return "Equal";**

**}**

**return "Not Equal";**

**}**

**compareEquality(10, "10");**

**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 falsecuando la comparación de igualdad devuelva truey viceversa . Al igual que el operador de igualdad, el operador de desigualdad descubrió los tipos de datos mientras los compara.

**Ejemplos**

**1 != 2 // true**

**1 != "1" // false**

**1 != '1' // false**

**1 != true // false**

**0 != false // false**

**// Configuración**

**function testNotEqual(val) {**

**if (val != 99) { // Cambia esta línea**

**return "Not Equal";**

**}**

**return "Equal";**

**}**

**testNotEqual(10);**

**Comparación con el operador de estricta desigualdad**

El operador de desigualdad !==es el opuesto lógico del operador de estricta igualdad. Esto significa "Estrictamente Desigual", y devuelve falsecuando la comparación de estricta igualdad devolvería truey viceversa . El operador de desigualdad no cambiará los tipos de datos.

**Ejemplos**

**3 !== 3 // false**

**3 !== '3' // true**

**4 !== 3 // true**

**// Configuración**

**function testStrictNotEqual(val) {**

**if (val !== 17) { // Cambia esta línea**

**return "Not Equal";**

**}**

**return "Equal";**

**}**

**testStrictNotEqual(10);**

**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 cambió los tipos de datos de valores mientras los compara.

**Ejemplos**

**5 > 3 // true**

**7 > '3' // true**

**2 > 3 // false**

**'1' > 9 // false**

**function testGreaterThan(val) {**

**if (val > 100 ) { // Cambia esta línea**

**return "Over 100";**

**}**

**if (val > 10 ) { // Cambia esta línea**

**return "Over 10";**

**}**

**return "10 or Under";**

**}**

**testGreaterThan(10);**

**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 descubrió los tipos de datos mientras los compara.

**Ejemplos**

**6 >= 6 // true**

**7 >= '3' // true**

**2 >= 3 // false**

**'7' >= 9 // false**

**function testGreaterOrEqual(val) {**

**if (val >= 20) { // Cambia esta línea**

**return "20 or Over";**

**}**

**if (val >= 10) { // Cambia esta línea**

**return "10 or Over";**

**}**

**return "Less than 10";**

**}**

**testGreaterOrEqual(10);**

**Comparación con el operador "menor que"**

El operador menor que ( <) compara los valores de dos números. Si el número a la izquierda es menor que el número a la derecha, devuelve true. De lo contrario, devuelve false. Al igual que el operador de igualdad, el operador menor que cambió los tipos de datos mientras los compara.

**Ejemplos**

**2 < 5 // true**

**'3' < 7 // true**

**5 < 5 // false**

**3 < 2 // false**

**'8' < 4 // false**

**function testLessThan(val) {**

**if (val < 25) { // Cambia esta línea**

**return "Under 25";**

**}**

**if (val < 55) { // Cambia esta línea**

**return "Under 55";**

**}**

**return "55 or Over";**

**}**

**testLessThan(10);**

**Comparacion 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 descubrió los tipos de datos mientras los compara.

**Ejemplos**

**4 <= 5 // true**

**'7' <= 7 // true**

**5 <= 5 // true**

**3 <= 2 // false**

**'8' <= 4 // false**

**function testLessOrEqual(val) {**

**if (val <= 12) { // Cambia esta línea**

**return "Smaller Than or Equal to 12";**

**}**

**if (val <= 24) { // Cambia esta línea**

**return "Smaller Than or Equal to 24";**

**}**

**return "More Than 24";**

**}**

**testLessOrEqual(10);**

**Comparaciones con el operador lógico "and"**

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

El mismo efecto podría lograrse anidando una ifdeclaración dentro de otra if.

**if (num > 5) {**

**if (num < 10) {**

**return "Yes";**

**}**

**}**

**return "No";**

Este código devolverá Yessi numes mayor que 5y menor que 10. La misma lógica se puede escribir con el operador lógico y .

**if (num > 5 && num < 10) {**

**return "Yes";**

**}**

**return "No";**

**function testLogicalAnd(val) {**

**// Cambia solo el código debajo de esta línea**

**if (val <= 50 && val >=25) {**

**return "Yes";**

**}**

**// Cambia solo el código encima de esta línea**

**return "No";**

**}**

**testLogicalAnd(10);**

**Comparaciones con el operador lógico "or"**

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

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

El patrón a continuación debería parecerle familiar de los puntos de referencia anteriores.

**if (num > 10) {**

**return "No";**

**}**

**if (num < 5) {**

**return "No";**

**}**

**return "Yes";**

Este código devolverá Yessi numestá entre 5y 10( 5e 10incluido). La misma lógica se puede escribir con el operador lógico o .

**if (num > 10 || num < 5) {**

**return "No";**

**}**

**return "Yes";**

**function testLogicalOr(val) {**

**// Cambia solo el código debajo de esta línea**

**if (val < 10 || val > 20) {**

**return "Outside";**

**}**

**// Cambia solo el código encima de esta línea**

**return "Inside";**

**}**

**testLogicalOr(15);**

**Introducción a las sentencias "Else"**

Cuando la condición en una sentencia ifes 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";**

**}**

**function testElse(val) {**

**let result = "";**

**// Cambia solo el código debajo de esta línea**

**if (val > 5) {**

**result = "Bigger than 5";**

**}**

**else {**

**result = "5 or Smaller";**

**}**

**// Cambia solo el código encima de esta línea**

**return result;**

**}**

**testElse(4);**

**Introducción a las sentencias "Else If"**

Si tienes múltiples condiciones que necesitan ser resueltas, puedes encadenar sentencias ifjunto 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";**

**}**

**function testElseIf(val) {**

**if (val > 10) {**

**return "Greater than 10";**

**}**

**else if (val < 5) {**

**return "Smaller than 5";**

**} else {**

**return "Between 5 and 10";**

**}**

**}**

**testElseIf(7);**

**Orden lógico en declaraciones If Else**

**El orden es importante en ifsus else ifdeclaraciones.**

La función se ejecuta de arriba hacia abajo, por lo que deberá tener cuidado con la declaración que aparece primero.

Tome estas dos funciones como ejemplo.

Aquí está el primero:

**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 el segundo simplemente cambia el orden de las declaraciones:

**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";**

**}**

**}**

Si bien estas dos funciones parecen casi idénticas, si les pasamos un número a ambas, obtenemos resultados diferentes.

**foo(0)**

**bar(0)**

foo(0)devolverá la cadena Less than oney bar(0)devolverá la cadena Less than two.

**function orderMyLogic(val) {**

**if (val < 5) {**

**return "Less than 5";**

**} else if (val < 10) {**

**return "Less than 10";**

**} else {**

**return "Greater than or equal to 10";**

**}**

**}**

**orderMyLogic(3);**

**Encadena sentencias if else**

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

**if (condition1) {**

**statement1**

**} else if (condition2) {**

**statement2**

**} else if (condition3) {**

**statement3**

**. . .**

**} else {**

**statementN**

**}**

**function testSize(num) {**

**// Only change code below this line**

**if (num < 5) {**

**return "Tiny";**

**} else if (num < 10) {**

**return "Small";**

**} else if (num < 15) {**

**return "Medium";**

**} else if (num < 20) {**

**return "Large";**

**} else {**

**return "Huge";**

**}**

**// Only change code above this line**

**}**

**console.log(testSize(20));**

**código de golf**

En el juego de Golf, cada agujero tiene un par, que significa, el número promedio de strokesque se espera que haga un golfista para hundir la pelota en el agujero para completar el juego. dependiendo de qué tan por encima o por debajo del parestán tus strokes, hay un nombre diferente.

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

**Golpes (golpes) devuelve**

**1 "¡Hoyo en uno!"**

**<= par - 2 "Águila"**

**par - 1 "Pajarito"**

**par "Par"**

**par + 1 "Espectro"**

**par + 2 "Doble bogey"**

**>= par + 3 "¡Vete a casa!"**

pary strokessiempre serán numéricos y positivos. Hemos añadido un arreglo de todos los nombres para tu conveniencia.

**var names = ["Hole-in-one!", "Eagle", "Birdie", "Par", "Bogey", "Double Bogey", "Go Home!"];**

**function golfScore(par, strokes) {**

**if (strokes == 1) {**

**return names[0]**

**} else if (strokes <= par - 2) {**

**return names[1]**

**} else if (strokes == par - 1) {**

**return names[2]**

**} else if (strokes == par) {**

**return names[3]**

**} else if (strokes == par + 1) {**

**return names[4]**

**} else if (strokes == par + 2) {**

**return names[5]**

**} else if (strokes >= par + 3) {**

**return names[6]**

**}**

**}**

**// Change these values to test**

**console.log(golfScore(5, 8));**

**Seleccionando entre muchas opciones con declaración switch**

Si necesita hacer coincidir un valor con muchas opciones, puede usar una declaración de cambio . Una switchdeclaración compara el valor con las declaraciones de caso que definen varios valores posibles. Cualquier instrucción de JavaScript válida se puede ejecutar dentro de un bloque de casos y se ejecutará desde el primer casevalor coincidente hasta que breakse encuentre a.

**Aquí hay un ejemplo de una declaración switch:**

**switch (fruit) {**

**case "apple":**

**console.log("The fruit is an apple");**

**break;**

**case "orange":**

**console.log("The fruit is an orange");**

**break;**

**}**

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

**function caseInSwitch(val) {**

**var answer = "";**

**switch(val) {**

**case 1:**

**answer = "alpha";**

**break;**

**case 2:**

**answer = "beta";**

**break;**

**case 3:**

**answer = "gamma";**

**break;**

**case 4:**

**answer = "delta";**

**break;**

**}**

**return answer;**

**}**

**// Change this value to test**

**console.log(caseInSwitch(3));**

**function caseInSwitch(val) {**

**let answer = "";**

**// Cambia solo el código debajo de esta línea**

**switch(val) {**

**case 1:**

**answer = "alpha";**

**break;**

**case 2:**

**answer = "beta";**

**break;**

**case 3:**

**answer = "gamma";**

**break;**

**case 4:**

**answer = "delta";**

**break;**

**}**

**// Cambia solo el código encima de esta línea**

**return answer;**

**}**

**caseInSwitch(1);**

**Agrega una opción predeterminada en las declaraciones switch**

En una declaración switchpuede que no seas capaz de especificar todos los valores posibles como declaraciones de case(caso). En su lugar, se puede agregar la declaración default, la cual se ejecutará si no se encontrarán declaraciones case. Piensa en ella como la última sentencia elseen una cadena if/else.

**Una declaración defaultdebe ser el último caso.**

**switch (num) {**

**case value1:**

**statement1;**

**break;**

**case value2:**

**statement2;**

**break;**

**...**

**default:**

**defaultStatement;**

**break;**

**}**

**function switchOfStuff(val) {**

**let answer = "";**

**switch (val) {**

**case "a":**

**answer = "apple";**

**break;**

**case "b":**

**answer = "bird";**

**break;**

**case "c":**

**answer = "cat";**

**break;**

**default:**

**answer = "stuff";**

**}**

**return answer;**

**}**

**switchOfStuff(1);**

**Múltiples opciones idénticas en las declaraciones "switch"**

Si la sentencia breakes omitida en un caso ( case) de una sentencia switch, las siguientes sentencias caseserán ejecutadas hasta encontrar un break. Si tienes múltiples entradas con la misma salida, puedes representarlas en una sentencia switchcomo esta:

**let result = "";**

**switch (val) {**

**case 1:**

**case 2:**

**case 3:**

**result = "1, 2, or 3";**

**break;**

**case 4:**

**result = "4 alone";**

**}**

**Los casos 1, 2 y 3 producirán el mismo resultado.**

**function sequentialSizes(val) {**

**var answer = "";**

**switch(val) {**

**case 1:**

**case 2:**

**case 3:**

**answer = "Low";**

**break;**

**case 4:**

**case 5:**

**case 6:**

**answer = "Mid";**

**break;**

**case 7:**

**case 8:**

**case 9:**

**answer = "High";**

**break;**

**}**

**return answer;**

**}**

**// Change this value to test**

**console.log(sequentialSizes(8));**

**Reemplazando cadenas de "If Else" por "Switch"**

Si tienes muchas opciones entre las que elegir, una sentencia switchpuede ser mas facil de escribir que muchas sentencias if/ else ifencadenadas. Lo siguiente:

**if (val === 1) {**

**answer = "a";**

**} else if (val === 2) {**

**answer = "b";**

**} else {**

**answer = "c";**

**}**

**puede reemplazarse por:**

**switch (val) {**

**case 1:**

**answer = "a";**

**break;**

**case 2:**

**answer = "b";**

**break;**

**default:**

**answer = "c";**

**}**

**function chainToSwitch(val) {**

**var answer = "";**

**// Only change code below this line**

**switch(val) {**

**case "bob":**

**answer = "Marley";**

**break;**

**case 42:**

**answer = "The Answer";**

**break;**

**case 1:**

**answer = "There is no #1";**

**break;**

**case 99:**

**answer = "Missed me by this much!";**

**break;**

**case 7:**

**answer = "Ate Nine";**

**break;**

**}**

**// Only change code above this line**

**return answer;**

**}**

**Devuelve valores booleanos desde funciones**

Puede recordar de la comparación con el operador de igualdad que todos los operadores de comparación devuelven un valor booleano trueo false.

A veces la gente usa una sentencia if/elsepara hacer una comparacion, como esta:

**function isEqual(a, b) {**

**if (a === b) {**

**return true;**

**} else {**

**return false;**

**}**

**}**

**Pero hay una mejor manera de hacer esto. Puesto que ===devuelve trueo false, podemos devolver el resultado de la comparación:**

**function isEqual(a, b) {**

**return a === b;**

**}**

**function isLess(a, b) {**

**// Cambia solo el código debajo de esta línea**

**return a < b;**

**// Cambia solo el código encima de esta línea**

**}**

**isLess(10, 15);**

**Patrón de devolución anticipado para funciones**

Cuando se alcanza una sentencia return, la ejecución de la función actual se detiene y el control se devuelve a la ubicación de la llamada.

**ejemplo**

**function myFun() {**

**console.log("Hello");**

**return "World";**

**console.log("byebye")**

**}**

**myFun();**

Lo anterior mostrará la cadena Helloen la consola y devolverá la cadena World. La cadena byebyenunca se muestra en la consola, porque la función termina en la sentencia return.

**// Setup**

**function abTest(a, b) {**

**// Only change code below this line**

**if (a < 0 || b < 0) {**

**return undefined;**

**}**

**// Only change code above this line**

**return Math.round(Math.pow(Math.sqrt(a) + Math.sqrt(b), 2));**

**}**

**// Change values below to test your code**

**console.log(abTest(-2,2));**

**Conteo de cartas**

En el juego de casino Blackjack, un jugador puede determinar si tiene una ventaja en la siguiente mano sobre la casa que lleva la cuenta del número relativo de cartas altas y bajas que quedan en la baraja. Esto se llama cuenta de tarjetas.

Tener más cartas altas en la barra es una ventaja para el jugador. Se le asigna un valor a cada carta de acuerdo a la siguiente tabla. Cuando el conteo es positivo, el jugador debería apostar alto. Cuando el conteo da 0 o negativo, el jugador debería apostar bajo.

**Cambios del conteo Cartas**

**+1 2, 3, 4, 5, 6**

**0 7, 8, 9**

**-1 10, 'J', 'Q', 'K', 'A'**

Escribirás una función para contar cartas. Recibirá un parámetro card(carta) que puede ser un número o una cadena y aumentar o reducir la variable global count(conteo) de acuerdo al valor de la carta (observa la tabla). La función devolverá una cadena con el conteo actual y la cadena Bet(Apuesta) si el conteo es positivo, o Hold(Espera) si el conteo es cero o negativo. El conteo actual y la decisión del jugador ( Beto Hold) deben estar separados por un solo espacio.

**Resultados de ejemplo: -3 Hold o5 Bet**

**Sugerencia**

**NO reinicia counta 0 cuando el valor sea 7, 8 o 9. NO devuelvas un arreglo.**

**NO incluye comillas (individuales o dobles) en el resultado.**

**var count = 0;**

**function cc(card) {**

**switch(card) {**

**case 2:**

**case 3:**

**case 4:**

**case 5:**

**case 6:**

**count++;**

**break;**

**case 10:**

**case "J":**

**case "Q":**

**case "K":**

**case "A":**

**count--;**

**break;**

**}**

**var holdbet = 'Hold'**

**if (count > 0) {**

**holdbet = 'Bet'**

**}**

**return count + " " + holdbet;**

**}**

**cc(2); cc(3); cc(7); cc('K'); cc('A');**

**console.log(cc(4))**

**Construir objetos de JavaScript**

Es posible que haya escuchado el término objectantes.

Los objetos son similares a arrays, excepto que en lugar de usar índices para acceder y modificar sus datos, accede a los datos en los objetos a través de lo que se llama properties.

Los objetos son útiles para almacenar datos de forma estructurada y pueden representar objetos del mundo real, como un gato.

Aquí hay un objeto gato de muestra:

**const cat = {**

**"name": "Whiskers",**

**"legs": 4,**

**"tails": 1,**

**"enemies": ["Water", "Dogs"]**

**};**

En este ejemplo, todas las propiedades se almacenan como cadenas, como name, legsy tails. Sin embargo, también puede usar números como propiedades. Incluso puede omitir las comillas para las propiedades de cadena de una sola palabra, de la siguiente manera:

**const anotherObject = {**

**make: "Ford",**

**5: "five",**

**"model": "focus"**

**};**

Sin embargo, si su objeto tiene propiedades que no son cadenas, JavaScript las encasillará automáticamente como cadenas.

**const myDog = {**

**// Only change code below this line**

**"name" : "string",**

**"legs" : 4,**

**"tails" : 1,**

**"friends" : []**

**// Only change code above this line**

**};**

**Acceder a propiedades de objetos con notación de puntos**

Hay dos maneras de acceder a las propiedades de un objeto: notación de puntos ( .) y notación de corchetes ( []), similar a un arreglo.

La notación de puntos es lo que se usa cuando conoce el nombre de la propiedad a la que intenta acceder con antelación.

Aquí hay un ejemplo de cómo usar la notación de puntos ( .) para leer la propiedad de un objeto:

**const myObj = {**

**prop1: "val1",**

**prop2: "val2"**

**};**

**const prop1val = myObj.prop1;**

**const prop2val = myObj.prop2;**

prop1valtendrá una cadena con valor val1y prop2valtendrá una cadena con valor val2.

**// Setup**

**const testObj = {**

**"hat": "ballcap",**

**"shirt": "jersey",**

**"shoes": "cleats"**

**};**

**// Only change code below this line**

**const hatValue = testObj.hat; // Change this line**

**const shirtValue = testObj.shirt; // Change this line**

**Acceso a propiedades de objetos con notación de corchetes**

La segunda forma de acceder a las propiedades de un objeto es la notación de corchetes ( []). Si la propiedad del objeto al que intenta acceder tiene un espacio en su nombre, deberá usar la notación de corchetes.

Sin embargo, aún puede usar la notación de corchetes en las propiedades del objeto sin espacios.

Aquí hay una muestra del uso de la notación de corchetes para leer la propiedad de un objeto:

**const myObj = {**

**"Space Name": "Kirk",**

**"More Space": "Spock",**

**"NoSpace": "USS Enterprise"**

**};**

**myObj["Space Name"];**

**myObj['More Space'];**

**myObj["NoSpace"];**

**myObj["Space Name"]sería la cadena Kirk, myObj['More Space']sería la cadena Spocky myObj["NoSpace"]sería la cadena USS Enterprise.**

Tenga en cuenta que los nombres de propiedad con espacios en ellos deben estar entre comillas (simples o dobles).

**// Setup**

**const testObj = {**

**"an entree": "hamburger",**

**"my side": "veggies",**

**"the drink": "water"**

**};**

**// Only change code below this line**

**const entreeValue = testObj["an entree"]; // Change this line**

**const drinkValue = testObj["the drink"]; // Change this line**

**Acceso a propiedades de objetos con variables**

Otro uso de la notación de corchetes en objetos es acceder a una propiedad que se almacena como el valor de una variable. Esto puede ser muy útil para iterar a través de las propiedades de un objeto o al acceder a una tabla de búsqueda.

Aquí hay un ejemplo del uso de una variable para acceder a una propiedad:

**const dogs = {**

**Fido: "Mutt",**

**Hunter: "Doberman",**

**Snoopie: "Beagle"**

**};**

**const myDog = "Hunter";**

**const myBreed = dogs[myDog];**

**console.log(myBreed);**

**La cadena Dobermanse mostraría en la consola.**

Tenga en cuenta que no usamos comillas alrededor del nombre de la variable cuando lo usamos para acceder a la propiedad porque estamos usando el valor de la variable, no el nombre .

**// Setup**

**const testObj = {**

**12: "Namath",**

**16: "Montana",**

**19: "Unitas"**

**};**

**// Only change code below this line**

**const playerNumber = 16; // Change this line**

**const player = testObj[playerNumber]; // Change this line**

**Actualización de propiedades de objetos**

Una vez que haya creado un objeto de JavaScript, puede actualizar sus propiedades en cualquier momento, tal como actualizaría cualquier otra variable. Puede usar la notación de puntos o corchetes para actualizar.

**Por ejemplo, veamos ourDog:**

**const ourDog = {**

**"name": "Camper",**

**"legs": 4,**

**"tails": 1,**

**"friends": ["everything!"]**

};

Ya que es un perro particularmente feliz, cambiemos su nombre a la cadena Happy Camper. Así es como actualizamos la propiedad de nombre de su objeto: ourDog.name = "Happy Camper";o ourDog["name"] = "Happy Camper";Ahora, cuando evaluamos ourDog.name, en lugar de obtener Camper, obtendremos su nuevo nombre, Happy Camper.

// Setup

const myDog = {

"name": "Coder",

"legs": 4,

"tails": 1,

"friends": ["freeCodeCamp Campers"]

};

// Only change code below this line

myDog.name = "Happy Coder";

**Agregar nuevas propiedades a un objeto de JavaScript**

Puede agregar nuevas propiedades a los objetos de JavaScript existentes de la misma manera que los modificaría.

Así es como agregaríamos una barkpropiedad a ourDog:

**ourDog.bark = "bow-wow";**

**o**

**ourDog["bark"] = "bow-wow";**

**Ahora, cuando evaluemos ourDog.bark, obtendremos su ladrido, bow-wow.**

**Ejemplo:**

**const ourDog = {**

**"name": "Camper",**

**"legs": 4,**

**"tails": 1,**

**"friends": ["everything!"]**

**};**

**ourDog.bark = "bow-wow";**

**const myDog = {**

**"name": "Happy Coder",**

**"legs": 4,**

**"tails": 1,**

**"friends": ["freeCodeCamp Campers"]**

**};**

**myDog.bark = "guau"**

**Eliminar propiedades de un objeto de JavaScript**

También podemos eliminar propiedades de objetos como este:

delete ourDog.bark;

Ejemplo:

**const ourDog = {**

**"name": "Camper",**

**"legs": 4,**

**"tails": 1,**

**"friends": ["everything!"],**

**"bark": "bow-wow"**

**};**

**delete ourDog.bark;**

Después de la última línea que se muestra arriba, ourDogse ve así:

**{**

**"name": "Camper",**

**"legs": 4,**

**"tails": 1,**

**"friends": ["everything!"]**

**}**

**// Setup**

**const myDog = {**

**"name": "Happy Coder",**

**"legs": 4,**

**"tails": 1,**

**"friends": ["freeCodeCamp Campers"],**

**"bark": "woof"**

**};**

**// Only change code below this line**

**delete myDog.tails;**

**Uso de objetos para búsquedas**

Los objetos se pueden considerar como un almacenamiento de clave/valor, como un diccionario. Si tiene datos tabulares, puede usar un objeto para buscar valores en lugar de una switchdeclaración o una if/elsecadena. Esto es más útil cuando sabe que sus datos de entrada están limitados a un cierto rango.

Este es un ejemplo de un objeto de artículo:

**const article = {**

**"title": "How to create objects in JavaScript",**

**"link": "https://www.freecodecamp.org/news/a-complete-guide-to-creating-objects-in-javascript-b0e2450655e8/",**

**"author": "Kaashan Hussain",**

**"language": "JavaScript",**

**"tags": "TECHNOLOGY",**

**"createdAt": "NOVEMBER 28, 2018"**

**};**

**const articleAuthor = article["author"];**

**const articleLink = article["link"];**

**const value = "title";**

**const valueLookup = article[value];**

articleAuthores la cadena Kaashan Hussain, articleLinkes la cadena https://www.freecodecamp.org/news/a-complete-guide-to-creating-objects-in-javascript-b0e2450655e8/y valueLookupes la cadena How to create objects in JavaScript.

**function phoneticLookup(val) {**

**let result = "";**

**var lookup = {**

**"alpha": "Adams",**

**"bravo": "Boston",**

**"charlie": "Chicago",**

**"delta": "Denver",**

**"echo": "Easy",**

**"foxtrot": "Frank"**

**};**

**result = lookup[val];**

**// Only change code above this line**

**return result;**

**}**

**phoneticLookup("charlie");**

**Probar objetos para propiedades**

A veces es útil verificar si la propiedad de un objeto dado existe o no. Podemos usar el .hasOwnProperty(propname)método de objetos para determinar si ese objeto tiene el nombre de propiedad dado. .hasOwnProperty()devoluciones trueo falsesi la propiedad se encuentra o no.

**Ejemplo**

**const myObj = {**

**top: "hat",**

**bottom: "pants"**

**};**

**myObj.hasOwnProperty("top");**

**myObj.hasOwnProperty("middle");**

**El primero hasOwnPropertyregresa true, mientras que el segundo regresa false.**

**function checkObj(obj, checkProp) {**

**if (obj.hasOwnProperty(checkProp)) {**

**return obj[checkProp];**

**} else {**

**return "Not Found";**

**}**

**}**

**Manipulación de objetos complejos**

A veces, es posible que desee almacenar datos en una estructura de datos flexible . Un objeto JavaScript es una forma de manejar datos flexibles. Permiten combinaciones arbitrarias de cadenas , números , valores booleanos , matrices , funciones y objetos .

He aquí un ejemplo de una estructura de datos compleja:

**const ourMusic = [**

**{**

**"artist": "Daft Punk",**

**"title": "Homework",**

**"release\_year": 1997,**

**"formats": [**

**"CD",**

**"Cassette",**

**"LP"**

**],**

**"gold": true**

**}**

**];**

Esta es una matriz que contiene un objeto dentro. El objeto tiene varios metadatos sobre un álbum. También tiene una formatsmatriz anidada. Si desea agregar más registros de álbumes, puede hacerlo agregando registros a la matriz de nivel superior. Los objetos contienen datos en una propiedad, que tiene un formato de clave-valor. En el ejemplo anterior, "artist": "Daft Punk"es una propiedad que tiene una clave de artisty un valor de Daft Punk.

Nota: Deberá colocar una coma después de cada objeto de la matriz, a menos que sea el último objeto de la matriz.

**var myMusic = [**

**{**

**"artist": "Billy Joel",**

**"title": "Piano Man",**

**"release\_year": 1973,**

**"formats": [**

**"CD",**

**"8T",**

**"LP"**

**],**

**"gold": true**

**},**

**{**

**artist: "Deep Purple",**

**title: "Smoke on the water",**

**release\_year: 1976,**

**formats: ["CD", "8T", "LP"]**

**}**

**];**

**Acceso a objetos anidados**

Se puede acceder a las subpropiedades de los objetos encadenando la notación de punto o corchete.

**Aquí hay un objeto anidado:**

**const ourStorage = {**

**"desk": {**

**"drawer": "stapler"**

**},**

**"cabinet": {**

**"top drawer": {**

**"folder1": "a file",**

**"folder2": "secrets"**

**},**

**"bottom drawer": "soda"**

**}**

**};**

**ourStorage.cabinet["top drawer"].folder2;**

**ourStorage.desk.drawer;**

**ourStorage.cabinet["top drawer"].folder2sería la cadena secrets, y ourStorage.desk.drawersería la cadena stapler.**

**// Setup**

**var myStorage = {**

**"car": {**

**"inside": {**

**"glove box": "maps",**

**"passenger seat": "crumbs"**

**},**

**"outside": {**

**"trunk": "jack"**

**}**

**}**

**};**

**var gloveBoxContents = myStorage.car.inside["glove box"]; // Change this line**

**console.log(gloveBoxContents)**

**Acceso a matrices anidadas**

Como hemos visto en ejemplos anteriores, los objetos pueden contener tanto objetos anidados como matrices anidadas. De manera similar al acceso a objetos anidados, la notación de corchete de matriz se puede encadenar para acceder a matrices anidadas.

**Aquí hay un ejemplo de cómo acceder a una matriz anidada:**

**const ourPets = [**

**{**

**animalType: "cat",**

**names: [**

**"Meowzer",**

**"Fluffy",**

**"Kit-Cat"**

**]**

**},**

**{**

**animalType: "dog",**

**names: [**

**"Spot",**

**"Bowser",**

**"Frankie"**

**]**

**}**

**];**

**ourPets[0].names[1];**

**ourPets[1].names[0];**

**ourPets[0].names[1]sería la cadena Fluffy, y ourPets[1].names[0]sería la cadena Spot.**

**const myPlants = [**

**{**

**type: "flowers",**

**list: [**

**"rose",**

**"tulip",**

**"dandelion"**

**]**

**},**

**{**

**type: "trees",**

**list: [**

**"fir",**

**"pine",**

**"birch"**

**]**

**}**

**];**

**var secondTree = myPlants[1].list[1];**

**Colección de registros**

Está creando una función que ayuda en el mantenimiento de una colección de álbumes musicales. La colección está organizada como un objeto que contiene varios álbumes que también son objetos. Cada álbum está representado en la colección con un único idcomo nombre de propiedad. Dentro de cada objeto de álbum, hay varias propiedades que describen información sobre el álbum. No todos los álbumes tienen información completa.

La updateRecordsfunción toma 4 argumentos representados por los siguientes parámetros de función:

**records- un objeto que contiene varios álbumes individuales**

**id- un número que representa un álbum específico en el recordsobjeto**

**prop- una cadena que representa el nombre de la propiedad del álbum para actualizar**

**value- una cadena que contiene la información utilizada para actualizar la propiedad del álbum**

Complete la función usando las reglas a continuación para modificar el objeto pasado a la función.

**Su función siempre debe devolver el recordsobjeto completo.**

**Si valuees una cadena vacía, elimine la proppropiedad dada del álbum.**

**Si propno es tracksy valueno es una cadena vacía, asigne el valueal de ese álbum prop.**

**Si propes tracksy el valor no es una cadena vacía, agregue al valuefinal de la matriz del álbum tracks. Primero debe crear esta matriz si el álbum no tiene una trackspropiedad.**

Nota:recordCollection Se utiliza una copia del objeto para las pruebas. No debe modificar directamente el recordCollectionobjeto.

**// Setup**

**const recordCollection = {**

**2548: {**

**albumTitle: 'Slippery When Wet',**

**artist: 'Bon Jovi',**

**tracks: ['Let It Rock', 'You Give Love a Bad Name']**

**},**

**2468: {**

**albumTitle: '1999',**

**artist: 'Prince',**

**tracks: ['1999', 'Little Red Corvette']**

**},**

**1245: {**

**artist: 'Robert Palmer',**

**tracks: []**

**},**

**5439: {**

**albumTitle: 'ABBA Gold'**

**}**

**};**

**// Only change code below this line**

**function updateRecords(records, id, prop, value) {**

**if (value === "") {**

**delete records[id][prop];**

**} else if (prop !== "tracks" && value !== "") {**

**records[id][prop] = value;**

**} else if (prop === "tracks" && value !== "" && records[id].hasOwnProperty("tracks") === false) {**

**records[id][prop] = [value];**

**} else if (prop === "tracks" && value !== "") {**

**records[id][prop].push(value);**

**}**

**return records;**

**}**

**updateRecords(recordCollection, 5439, 'artist', 'ABBA');**

**Iterar con bucles while de JavaScript**

Puede ejecutar el mismo código varias veces mediante un bucle.

El primer tipo de bucle que aprenderemos se llama bucle whileporque se ejecuta mientras una condición específica es verdadera y se detiene una vez que esa condición ya no es verdadera.

**const ourArray = [];**

**let i = 0;**

**while (i < 5) {**

**ourArray.push(i);**

**i++;**

**}**

**En el ejemplo de código anterior, el whilebucle se ejecutará 5 veces y agregará los números del 0 al 4 a ourArray.**

**Intentemos hacer funcionar un bucle while insertando valores en una matriz.**

**// Setup**

**const myArray = [];**

**// Only change code below this line.**

**let i = 5;**

**while(i >= 0) {**

**myArray.push(i);**

**i--;**

**}**

**Iterar con JavaScript For Loops**

Puede ejecutar el mismo código varias veces mediante un bucle.

El tipo más común de bucle de JavaScript se denomina forbucle porque se ejecuta una cantidad específica de veces.

Los bucles for se declaran con tres expresiones opcionales separadas por punto y coma:

for (a; b; c), donde aes la declaración de inicialización, bes la declaración de condición y ces la expresión final.

La declaración de inicialización se ejecuta una sola vez antes de que comience el ciclo. Por lo general, se usa para definir y configurar su variable de bucle.

La declaración de la condición se evalúa al comienzo de cada iteración del ciclo y continuará mientras se evalúe como true. Cuando la condición está falseal comienzo de la iteración, el bucle dejará de ejecutarse. Esto significa que si la condición comienza como falsa, su ciclo nunca se ejecutará.

La expresión final se ejecuta al final de cada iteración de ciclo, antes de la siguiente verificación de condición y, por lo general, se usa para incrementar o disminuir el contador de ciclo.

En el siguiente ejemplo, inicializamos con i = 0e iteramos mientras nuestra condición i < 5es verdadera. Incrementaremos ien 1cada iteración de bucle como i++nuestra expresión final.

**const ourArray = [];**

**for (let i = 0; i < 5; i++) {**

**ourArray.push(i);**

**}**

**ourArrayahora tendrá el valor [0, 1, 2, 3, 4].**

**// Setup**

**const myArray = [];**

**// Only change code below this line**

**for (var i = 1; i < 6; i++) {**

**myArray.push(i);**

**}**

**console.log(myArray);**

**Iterar números impares con un bucle For**

Los bucles for no tienen que iterar uno a la vez. Cambiando nuestro final-expression, podemos contar por números pares.

Comenzaremos en i = 0y repetiremos while i < 10. Incrementaremos ien 2 cada ciclo con i += 2.

**const ourArray = [];**

**for (let i = 0; i < 10; i += 2) {**

**ourArray.push(i);**

**}**

ourArrayahora contendrá [0, 2, 4, 6, 8]. Cambiemos nuestro initializationpara que podamos contar por números impares.

**// Setup**

**const myArray = [];**

**// Only change code below this line**

**for (var i = 1; i < 10; i += 2) {**

**myArray.push(i);**

**}**

**console.log(myArray);**

**Cuenta hacia atrás con un bucle "for"**

Un bucle for también puede contar hacia atrás, siempre que definamos las condiciones adecuadas.

Para poder disminuirle dos cada iteración, necesitaremos cambiar nuestra inicialización, condición, y expresión final.

Empezaremos en i = 10 e iteraremos mientras i > 0. Disminuiremos i en 2 por cada bucle con i -= 2.

**const ourArray = [];**

**for (let i = 10; i > 0; i -= 2) {**

**ourArray.push(i);**

**}**

ourArray ahora contendrá [10, 8, 6, 4, 2]. Ahora cambiemos el valor de inicialización y la expresión final de nuestro bucle para que podamos contar hacia atrás de dos en dos y así crear un arreglo descendente de números impares.

**// Configuración**

**const myArray = [];**

**// Cambia solo el código debajo de esta línea**

**for (var i = 9; i > 0; i -= 2) {**

**myArray.push(i);**

**}**

**console.log(myArray);**

**Itera a través de un arreglo con un bucle "for"**

**Una tarea común en JavaScript es iterar a través del contenido de un arreglo. Una forma de hacerlo es con un bucle for. Este código mostrará cada elemento del arreglo arr en la consola:**

**const arr = [10, 9, 8, 7, 6];**

**for (let i = 0; i < arr.length; i++) {**

**console.log(arr[i]);**

**}**

Recuerda que los arreglos tienen una indexación basada en cero, lo que significa que el último índice del arreglo es igual a su longitud menos uno (length - 1). Nuestra condición para este bucle es i < arr.length, que detiene el bucle cuando i es igual a length. En este caso, la última iteración es i === 4, es decir, cuando i se convierte en igual a arr.length - 1 y resultados 6 a la consola. Entonces i aumenta a 5, y el bucle termina i < arr.length es false.

**// Configuración**

**// Cambia solo el código debajo de esta línea**

**var myArr = [ 2, 3, 4, 5, 6];**

**var total = 0;**

**for (var i = 0; i < myArr.length; i++) {**

**total += myArr[i];**

**}**

**console.log(total);**

**Anida bucles "for"**

Si tienes un arreglo multidimensional, puedes utilizar la misma lógica que aprendimos anteriormente para recorrer tanto el arreglo como cualquier sub-arreglo. Aquí hay un ejemplo:

**const arr = [**

**[1, 2], [3, 4], [5, 6]**

**];**

**for (let i = 0; i < arr.length; i++) {**

**for (let j = 0; j < arr[i].length; j++) {**

**console.log(arr[i][j]);**

**}**

**}**

Esto imprime cada sub-elemento dentro de arr uno a la vez. Ten en cuenta que para el bucle interior, estamos comprobando el .length de arr[i], ya que arr[i] es en sí mismo es un arreglo.

**function multiplyAll(arr) {**

**var product = 1;**

**for (var i=0; i < arr.length; i++) {**

**for (var j=0; j < arr[i].length; j++) {**

**product \*= arr[i][j];**

**}**

**}**

**return product;**

**}**

**var product = multiplyAll([[1,2],[3,4],[5,6,7]]);**

**console.log(product);**

**Itera con el bucle "do...while" de JavaScript**

El siguiente tipo de bucle que aprenderás se llama bucle do...while. Se llama bucle do...whileporque primero hace ( do) una pasada por el código dentro del bucle sin importar qué, y luego continúa el bucle mientras ( while) la condición específica sea verdadera ( true).

**const ourArray = [];**

**let i = 0;**

**do {**

**ourArray.push(i);**

**i++;**

**} while (i < 5);**

El ejemplo anterior se comporta de forma similar a otros tipos de bucles, siendo el arreglo resultante [0, 1, 2, 3, 4]. Sin embargo, lo que hace que el bucle do...whilesea diferente a bucles es cómo se comporta cuando la condición falla en la primera verificación. Veamos esto en acción. Aquí hay un bucle whilenormal que ejecuta el código en el bucle mientras i < 5:

**const ourArray = [];**

**let i = 5;**

**while (i < 5) {**

**ourArray.push(i);**

**i++;**

**}**

En este ejemplo, inicializamos el valor de ourArrayun arreglo vacío y el valor de iun 5. Cuando ejecutamos el bucle while, la condición se evalúa como falseporque ino es inferior a 5, así que no ejecutamos el código dentro del bucle. El resultado es que ourArrayterminará sin valores añadidos, y todavía se verá como []una vez el código del ejemplo anterior haya terminado de ejecutarse. Ahora, dale un vistazo a un bucle do...while:

**const ourArray = [];**

**let i = 5;**

**do {**

**ourArray.push(i);**

**i++;**

**} while (i < 5);**

En este caso, inicializamos el valor de ia 5, tal como lo hicimos en el bucle while. Cuando lleguemos a la siguiente línea, no hay ninguna condición para evaluar, así que entramos al código dentro de las llaves y se ejecuta. Añadiremos un único elemento al arreglo y luego incrementaremos iantes de llegar a la verificación de la condición. Cuando finalmente evaluamos la condición i < 5en la última línea, vemos que el valor de ies ahora 6, por lo que falla la comprobación condicional. Salimos del bucle y hemos terminado. Al final del ejemplo anterior, el valor de ourArrayes [5]. Esencialmente, un bucle do...whileasegura que el código dentro del bucle se ejecuta al menos una vez. Intentemos construir un bucledo...whilepara que funcione empujando valores a un arreglo.

**// Configuración**

**const myArray = [];**

**let i = 10;**

**// Cambia solo el código debajo de esta línea**

**do {**

**myArray.push(i);**

**i++;**

**} while (i < 5)**

**Reemplaza bucles usando recursión**

La recursión es el concepto que una función puede expresarse en términos de sí misma. Para ayudar a comprender esto, comience pensando en la siguiente tarea: multiplicar los primeros nelementos de un arreglo para crear el producto de esos elementos. Usando un bucle for, puedes hacer esto:

**function multiply(arr, n) {**

**let product = 1;**

**for (let i = 0; i < n; i++) {**

**product \*= arr[i];**

**}**

**return product;**

**}**

Sin embargo, nota que multiply(arr, n) == multiply(arr, n - 1) \* arr[n - 1]. Esto significa que puedes reescribir multiplyen términos de sí mismo y que nunca necesitas hacer uso de un bucle.

**function multiply(arr, n) {**

**if (n <= 0) {**

**return 1;**

**} else {**

**return multiply(arr, n - 1) \* arr[n - 1];**

**}**

**}**

La versión recursiva de multiplyse desglosa así. En el caso base , donde n <= 0, devuelve 1. Para más valores grandes de n, se llama a sí mismo, pero con n - 1. Esa llamada de función se evalúa de la misma manera, llamando a multiplyotra vez hasta que n <= 0. En este punto, todas las funciones pueden devolver y la multiplyoriginal devuelve la respuesta.

Nota: Las funciones recursivas deben tener un caso base cuando devuelven sin tener que llamar a la función de nuevo (en este ejemplo, cuando n <= 0), de lo contrario nunca podrán terminar de ejecutarse.

**function sum(arr, n) {**

**// Cambia solo el código debajo de esta línea**

**if(n <= 0) {**

**return 0;**

**} else {**

**return sum(arr, n - 1) + arr[n - 1];**

**}**

**// Cambia solo el código encima de esta línea**

**}**

**Búsqueda de perfiles**

Tenemos un arreglo de objetos que representan a diferentes personas en nuestras listas de contactos.

Una función lookUpProfileque recibe nombre ( name) y una propiedad ( prop) como argumentos preescritos para ti.

La función debe verificar si el nombre ( name) es el nombre de pila del contacto ( firstName) y la propiedad ( prop) dada es una propiedad de ese contacto.

Si ambos son verdaderos, entonces devolverán el "valor" de esa propiedad.

Si nameno corresponde a ningún contacto, entonces devuelve la cadena No such contact.

Si propno corresponde a ninguna propiedad válida de un contacto encontrado que coincide con nameentonces devuelve la cadena No such property.

**//Setup**

**var contacts = [**

**{**

**"firstName": "Akira",**

**"lastName": "Laine",**

**"number": "0543236543",**

**"likes": ["Pizza", "Coding", "Brownie Points"]**

**},**

**{**

**"firstName": "Harry",**

**"lastName": "Potter",**

**"number": "0994372684",**

**"likes": ["Hogwarts", "Magic", "Hagrid"]**

**},**

**{**

**"firstName": "Sherlock",**

**"lastName": "Holmes",**

**"number": "0487345643",**

**"likes": ["Intriguing Cases", "Violin"]**

**},**

**{**

**"firstName": "Kristian",**

**"lastName": "Vos",**

**"number": "unknown",**

**"likes": ["JavaScript", "Gaming", "Foxes"]**

**}**

**];**

**function lookUpProfile(name, prop){**

**for (var i = 0; i < contacts.length; i++) {**

**if(contacts[i].firstName === name) {**

**return contacts[i][prop] || "No such property";**

**}**

**}**

**return "No such contact";**

**}**

**// Change these values to test your function**

**var data = lookUpProfile("Sherlock", "hello");**

**console.log(data);**

**Genera fracciones aleatorias con JavaScript**

Los números aleatorios son útiles para crear comportamientos aleatorios.

JavaScript tiene una función Math.random()que genera un número decimal aleatorio entre 0(inclusivo) y 1(exclusivo). Así que Math.random()puede devolver un 0pero nunca devuelve un 1.

Nota: Al igual que el almacenamiento de valores con el operador de protección , todas las llamadas a la función se resolverán antes de que se ejecute el return, por lo que el returndevuelve el valor de la función Math.random().

**function randomFraction() {**

**// Cambia solo el código debajo de esta línea**

**return Math.random();**

**// Cambia solo el código encima de esta línea**

**}**

**Genera numeros enteros aleatorios con JavaScript**

Es genial que podamos generar numeros decimales aleatorios, pero es incluso mas util si lo usamos para generar numeros enteros aleatorios.

Usa Math.random()para generar un decimal aleatorio.

Multiplica ese decimal ocasionalmente por 20.

Utilice otra función, Math.floor()para redondear el número hacia abajo a su número entero más cercano.

Recuerda que Math.random()nunca devolverá un 1y porque estamos redondeando hacia abajo, es imposible obtener 20. Esta técnica nos dará un número entero entre 0y 19.

Poniendo todo junto, así es como se ve nuestro código:

**Math.floor(Math.random() \* 20);**

Estamos llamando a Math.random(), multiplicando el resultado por 20 y pasando el valor a la función Math.floor()para redondear el valor hacia abajo al número entero más cercano.

**var randomNumberBetween0and19 = Math.floor(Math.random() \* 20);**

**function randomWholeNum() {**

**return Math.floor(Math.random() \* 10);**

**}**

**console.log(randomWholeNum());**

**Genera numeros enteros aleatorios dentro de un rango**

En lugar de generar un número entero aleatorio entre cero y un número dado como lo hicimos anteriormente, podemos generar un número entero aleatorio que se encuentra dentro de un rango de dos números específicos.

Para ello, definiremos un número mínimo miny un número máximo max.

Esta es la fórmula que utilizaremos. Tómate un momento para leerla e intenta entender lo que este código está haciendo:

**Math.floor(Math.random() \* (max - min + 1)) + min**

**function randomRange(myMin, myMax) {**

**return Math.floor(Math.random() \* (myMax - myMin + 1)) + myMin;**

**}**

**var myRandom = randomRange(5, 15);**

**console.log(myRandom);**

**Utiliza la función "parseInt"**

La función parseInt()analiza una cadena y devuelve un entero. A continuación, le presentamos un ejemplo:

**const a = parseInt("007");**

La función anterior convierte la cadena 007al entero 7. Si el primer personaje de la cadena no puede ser convertido en un número, entonces devuelve NaN.

**function convertToInteger(str) {**

**return parseInt(str);**

**}**

**convertToInteger("56");**

**Utiliza la función "parseInt" con Radix (Base)**

La función parseInt()analiza una cadena y devuelve un entero. Recibe un segundoargumento para la base (radix), que especifica la base del número representado en la cadena. La base (radix) puede ser un número entero entre 2 y 36.

La llamada a la función se realiza de la siguiente manera:

**parseInt(string, radix);**

Y aquí hay un ejemplo:

**const a = parseInt("11", 2);**

La variable radix indica que 11está en el sistema binario, o base 2. Este ejemplo convierte la cadena 11a un entero 3.

**function convertToInteger(str) {**

**return parseInt(str, 2)**

**}**

**convertToInteger("10011");**

**Usa el operador condicional (ternario)**

El operador condicional , también llamado el operador ternario , puede usarse como una expresión if-else de una sola línea.

La sintaxis es a ? b : c, donde aes la condición, bes el código a ejecutar cuando la condición devuelve true, y ces el código a ejecutar cuando la condición devuelve false.

La siguiente función utiliza una sentencia if/elsepara comprobar una condición:

**function findGreater(a, b) {**

**if(a > b) {**

**return "a is greater";**

**}**

**else {**

**return "b is greater or equal";**

**}**

**}**

Esto puede reescribirse usando el operador condicional:

**function findGreater(a, b) {**

**return a > b ? "a is greater" : "b is greater or equal";**

**}**

**function checkEqual(a, b) {**

**return a === b ? "Equal" : "Not Equal";**

**}**

**checkEqual(1, 2);**

**Usa múltiples operadores condicionales (ternarios)**

En el desafío anterior, usaste un único operador condicional. También puedes encadenarlos para comprobar múltiples condiciones.

La siguiente función utiliza sentencias if, else if, y elsepara comprobar múltiples condiciones:

**function findGreaterOrEqual(a, b) {**

**if (a === b) {**

**return "a and b are equal";**

**}**

**else if (a > b) {**

**return "a is greater";**

**}**

**else {**

**return "b is greater";**

**}**

**}**

La función anterior puede ser reescrita utilizando múltiples operadores condicionales:

**function findGreaterOrEqual(a, b) {**

**return (a === b) ? "a and b are equal"**

**: (a > b) ? "a is greater"**

**: "b is greater";**

**}**

Se considera una buena práctica dar formato a múltiples operadores condicionales de forma que cada condición esté en una línea separada, como se muestra arriba. Usar múltiples operadores condicionales sin una sangría adecuada puede hacer tu código difícil de leer. Por ejemplo:

**function findGreaterOrEqual(a, b) {**

**return (a === b) ? "a and b are equal" : (a > b) ? "a is greater" : "b is greater";**

**}**

**function checkSign(num) {**

**return num > 0 ? "positive" : num < 0 ? "negative" : "zero"**

**}**

**console.log(checkSign(0));**

**Utiliza recursion para crear una cuenta regresiva**

En un reto anterior , aprendiste a utilizar la recursividad para sustituir un bucle for. Ahora, echamos un vistazo a una función más compleja que devuelve un arreglo de enteros consecutivos comenzando con 1hasta el número pasado a la función.

Como se menciona en el desafío anterior, habrá un caso base . El caso base le dice a la función recursiva cuando no necesita llamarse a sí mismo. Es un caso simple donde el valor de retorno ya se conoce. También habrá una llamada recursiva la cual ejecuta la función original con argumentos diferentes. Si la función se escribe correctamente, eventualmente se logrará la base del caso.

Por ejemplo, digamos que quieres escribir una función recursiva que devuelva un arreglo conteniendo los números 1hasta n. Esta función puede aceptar un argumento, nque representa el número final. Entonces llamarse a sí mismo con valores progresivamente más pequeños de nhasta que alcance 1. Podrías escribir la función de la siguiente manera:

**function countup(n) {**

**if (n < 1) {**

**return [];**

**} else {**

**const countArray = countup(n - 1);**

**countArray.push(n);**

**return countArray;**

**}**

**}**

**console.log(countup(5));**

El valor [1, 2, 3, 4, 5]se muestra en la consola.

Al principio, esto parece contraintuitivo ya que el valor decrece n , pero los valores en el arreglo final se están incrementando . Esto sucede porque la inserción ocurre al último, después de la llamada recursiva. En el punto donde nes empujado en el arreglo, countup(n - 1)ya ha sido evaluado y devuelto [1, 2, ..., n - 1].

**function countdown(n) {**

**if (n < 1) {**

**return [];**

**} else {**

**const arr = countdown(n - 1);**

**arr.unshift(n);**

**return arr;**

**}**

**}**

**Usa recursión para crear un rango de números**

Continuando con el desafío anterior, te ofrecemos otra oportunidad de crear una función recursiva para resolver un problema.

**function rangeOfNumbers(startNum, endNum) {**

**if (endNum < startNum) {**

**return [];**

**} else {**

**const numbers = rangeOfNumbers(startNum, endNum - 1);**

**numbers.push(endNum);**

**return numbers;**

**}**

**}**