**Estructuras de datos básicos**

Usar una matriz para almacenar una colección de datos

El siguiente es un ejemplo de la implementación más simple de una estructura de datos de matriz. Esto se conoce como una matriz unidimensional , lo que significa que solo tiene un nivel o que no tiene ninguna otra matriz anidada dentro de ella. Tenga en cuenta que contiene valores booleanos , cadenas y números , entre otros tipos de datos de JavaScript válidos:

**let simpleArray = ['one', 2, 'three', true, false, undefined, null];**

**console.log(simpleArray.length);**

Se muestra la console.logllamada 7.

Todas las matrices tienen una propiedad de longitud, que, como se muestra arriba, se puede acceder muy fácilmente con la sintaxis Array.length. A continuación se puede ver una implementación más compleja de una matriz. Esto se conoce como una matriz multidimensional o una matriz que contiene otras matrices. Tenga en cuenta que esta matriz también contiene objetos de JavaScript , que examinaremos muy de cerca en la siguiente sección, pero por ahora, todo lo que necesita saber es que las matrices también pueden almacenar objetos complejos.

**let complexArray = [**

**[**

**{**

**one: 1,**

**two: 2**

**},**

**{**

**three: 3,**

**four: 4**

**}**

**],**

**[**

**{**

**a: "a",**

**b: "b"**

**},**

**{**

**c: "c",**

**d: "d"**

**}**

**]**

**];**

**let yourArray = ["abc" , 5, true, 4, false]; // Change this line**

**Acceda al contenido de una matriz usando la notación de paréntesis**

La característica fundamental de cualquier estructura de datos es, por supuesto, la capacidad no solo de almacenar datos, sino también de poder recuperar esos datos a pedido. Entonces, ahora que hemos aprendido cómo crear una matriz, comencemos a pensar en cómo podemos acceder a la información de esa matriz.

Cuando definimos una matriz simple como se ve a continuación, hay 3 elementos en ella:

**let ourArray = ["a", "b", "c"];**

En una matriz, cada elemento de la matriz tiene un índice . Este índice se duplica como la posición de ese elemento en la matriz y cómo se hace referencia a él. Sin embargo, es importante tener en cuenta que las matrices de JavaScript están indexadas a cero , lo que significa que el primer elemento de una matriz está realmente en la posición cero , no en la primera. Para recuperar un elemento de una matriz, podemos encerrar un índice entre paréntesis y agregarlo al final de una matriz, o más comúnmente, a una variable que hace referencia a un objeto de matriz. Esto se conoce como notación de paréntesis . Por ejemplo, si queremos recuperar el afrom ourArrayy asignarlo a una variable, podemos hacerlo con el siguiente código:

**let ourVariable = ourArray[0];**

Ahora ourVariabletiene el valor de a.

Además de acceder al valor asociado con un índice, también puede establecer un índice en un valor utilizando la misma notación:

**ourArray[1] = "not b anymore";**

Usando la notación de paréntesis, ahora hemos restablecido el elemento en el índice 1 de la cadena ba not b anymore. Ahora ourArrayes ["a", "not b anymore", "c"]\_

**let myArray = ["a", "b", "c", "d"];**

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

**myArray[1] = 7**

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

**console.log(myArray);**

**Agregue elementos a una matriz con push() y unshift()**

La longitud de una matriz, como los tipos de datos que puede contener, no es fija. Las matrices se pueden definir con una longitud de cualquier número de elementos, y los elementos se pueden agregar o eliminar con el tiempo; en otras palabras, las matrices son mutables . En este desafío, veremos dos métodos con los que podemos modificar una matriz mediante programación: Array.push()y Array.unshift().

Ambos métodos toman uno o más elementos como parámetros y agregan esos elementos a la matriz a la que se llama el método; el push()método agrega elementos al final de una matriz y unshift()agrega elementos al principio. Considera lo siguiente:

**let twentyThree = 'XXIII';**

**let romanNumerals = ['XXI', 'XXII'];**

**romanNumerals.unshift('XIX', 'XX');**

romanNumeralstendría el valor ['XIX', 'XX', 'XXI', 'XXII'].

**romanNumerals.push(twentyThree);**

romanNumeralstendría el valor ['XIX', 'XX', 'XXI', 'XXII', 'XXIII']. Tenga en cuenta que también podemos pasar variables, lo que nos permite una mayor flexibilidad para modificar dinámicamente los datos de nuestra matriz.

**function mixedNumbers(arr) {**

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

**arr.unshift("I", 2, "three");**

**arr.push(7, "VIII", 9);**

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

**return arr;**

**}**

**console.log(mixedNumbers(['IV', 5, 'six']));**

**Eliminar elementos de una matriz con pop() y shift()**

Ambos push()y unshift()tienen métodos correspondientes que son casi opuestos funcionales: pop()y shift(). Como ya habrá adivinado, en lugar de agregar, pop() elimina un elemento del final de una matriz, mientras que shift()elimina un elemento del principio. La diferencia clave entre pop()y shift()y sus primos push()y unshift(), es que ninguno de los métodos toma parámetros, y cada uno solo permite que una matriz sea modificada por un solo elemento a la vez.

Vamos a ver:

**let greetings = ['whats up?', 'hello', 'see ya!'];**

**greetings.pop();**

greetingstendría el valor ['whats up?', 'hello'].

**greetings.shift();**

greetingstendría el valor ['hello'].

También podemos devolver el valor del elemento eliminado con cualquier método como este:

**let popped = greetings.pop();**

greetingstendría el valor [], y poppedtendría el valor hello.

**function popShift(arr) {**

**let popped = arr.pop(); // Change this line**

**let shifted = arr.shift(); // Change this line**

**return [shifted, popped];**

**}**

**console.log(popShift(['challenge', 'is', 'not', 'complete']));**

**Eliminar elementos usando empalme ()**

Bien, hemos aprendido cómo eliminar elementos del principio y el final de los arreglos usando shift()y pop(), pero ¿qué pasa si queremos eliminar un elemento de algún punto intermedio? ¿O eliminar más de un elemento a la vez? Bueno, ahí es donde splice()entra en juego. splice()Nos permite hacer exactamente eso: eliminar cualquier número de elementos consecutivos de cualquier parte de una matriz.

splice()puede tomar hasta 3 parámetros, pero por ahora, nos centraremos solo en los primeros 2. Los dos primeros parámetros de splice()son números enteros que representan índices o posiciones de elementos en la matriz a la que splice()se está llamando. Y recuerde, los arreglos tienen un índice cero , por lo que para indicar el primer elemento de un arreglo, usaríamos 0. splice()El primer parámetro de representa el índice en la matriz desde la que comenzar a eliminar elementos, mientras que el segundo parámetro indica la cantidad de elementos que se eliminarán. Por ejemplo:

**let array = ['today', 'was', 'not', 'so', 'great'];**

**array.splice(2, 2);**

Aquí eliminamos 2 elementos, comenzando con el tercer elemento (en el índice 2). arraytendría el valor ['today', 'was', 'great'].

splice()no solo modifica la matriz a la que se llama, sino que también devuelve una nueva matriz que contiene el valor de los elementos eliminados:

**let array = ['I', 'am', 'feeling', 'really', 'happy'];**

**let newArray = array.splice(3, 2);**

newArraytiene el valor ['really', 'happy'].

**const arr = [2, 4, 5, 1, 7, 5, 2, 1];**

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

**arr.splice(1 ,4);**

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

**console.log(arr);**

**Agregar elementos usando empalme ()**

¿Recuerdas en el último desafío que mencionamos que splice()puede tomar hasta tres parámetros? Bueno, puede usar el tercer parámetro, compuesto por uno o más elementos, para agregarlo a la matriz. Esto puede ser increíblemente útil para cambiar rápidamente un elemento, o un conjunto de elementos, por otro.

**const numbers = [10, 11, 12, 12, 15];**

**const startIndex = 3;**

**const amountToDelete = 1;**

**numbers.splice(startIndex, amountToDelete, 13, 14);**

**console.log(numbers);**

La segunda aparición de 12se elimina y agregamos 13y 14en el mismo índice. La numbersmatriz ahora sería [ 10, 11, 12, 13, 14, 15 ].

Aquí, comenzamos con una serie de números. Luego, le pasamos lo siguiente a splice(): El índice en el que comenzar a eliminar elementos (3), el número de elementos a eliminar (1) y los argumentos restantes (13, 14) se insertarán a partir de ese mismo índice. Tenga en cuenta que puede haber cualquier número de elementos (separados por comas) después de amountToDelete, cada uno de los cuales se inserta.

**Copiar elementos de matriz usando slice()**

El siguiente método que cubriremos es slice(). En lugar de modificar una matriz, slice()copia o extrae una cantidad dada de elementos a una nueva matriz, dejando intacta la matriz a la que se llama. slice()toma solo 2 parámetros: el primero es el índice en el que comienza la extracción y el segundo es el índice en el que se detiene la extracción (la extracción ocurrirá hasta el elemento en este índice, pero sin incluirlo). Considera esto:

**let weatherConditions = ['rain', 'snow', 'sleet', 'hail', 'clear'];**

**let todaysWeather = weatherConditions.slice(1, 3);**

todaysWeathertendría el valor ['snow', 'sleet'], mientras que weatherConditionstodavía tendría ['rain', 'snow', 'sleet', 'hail', 'clear'].

En efecto, hemos creado una nueva matriz extrayendo elementos de una matriz existente.

**function forecast(arr) {**

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

**return arr.slice(2,4);**

**}**

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

**console.log(forecast(['cold', 'rainy', 'warm', 'sunny', 'cool', 'thunderstorms']));**

**Copie una matriz con el operador de propagación**

Si bien slice()nos permite ser selectivos sobre qué elementos de una matriz copiar, entre varias otras tareas útiles, el nuevo operador de extensión de ES6 nos permite copiar fácilmente todos los elementos de una matriz, en orden, con una sintaxis simple y fácil de leer. La sintaxis extendida simplemente se ve así:...

En la práctica, podemos usar el operador de propagación para copiar una matriz de esta manera:

**let thisArray = [true, true, undefined, false, null];**

**let thatArray = [...thisArray];**

thatArray[true, true, undefined, false, null]es igual thisArraypermanece sin cambios y thatArraycontiene los mismos elementos que thisArray.

**function copyMachine(arr, num) {**

**let newArr = [];**

**while (num >= 1) {**

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

**newArr.push([...arr]);**

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

**num--;**

**}**

**return newArr;**

**}**

**console.log(copyMachine([true, false, true], 2));**

**Combinar matrices con el operador de propagación**

Otra gran ventaja del operador de dispersión es la capacidad de combinar matrices, o de insertar todos los elementos de una matriz en otra, en cualquier índice. Con sintaxis más tradicionales, podemos concatenar arreglos, pero esto solo nos permite combinar arreglos al final de uno y al comienzo de otro. La sintaxis extendida hace que la siguiente operación sea extremadamente simple:

**let thisArray = ['sage', 'rosemary', 'parsley', 'thyme'];**

**let thatArray = ['basil', 'cilantro', ...thisArray, 'coriander'];**

thatArraytendría el valor ['basil', 'cilantro', 'sage', 'rosemary', 'parsley', 'thyme', 'coriander'].

Usando la sintaxis extendida, acabamos de lograr una operación que hubiera sido más compleja y detallada si hubiéramos usado métodos tradicionales.

**function spreadOut() {**

**let fragment = ['to', 'code'];**

**let sentence = ["learning", ...fragment, "is", "fun"]; // Change this line**

**return sentence;**

**}**

**console.log(spreadOut());**

**Comprobar la presencia de un elemento con indexOf()**

Dado que las matrices se pueden cambiar o mutar en cualquier momento, no hay garantía sobre dónde estará una parte de los datos en particular en una matriz determinada, o si ese elemento aún existe. Afortunadamente, JavaScript nos brinda otro método incorporado, indexOf()que nos permite verificar rápida y fácilmente la presencia de un elemento en una matriz. indexOf()toma un elemento como parámetro y, cuando se llama, devuelve la posición o el índice de ese elemento, o -1si el elemento no existe en la matriz.

Por ejemplo:

**let fruits = ['apples', 'pears', 'oranges', 'peaches', 'pears'];**

**fruits.indexOf('dates');**

**fruits.indexOf('oranges');**

**fruits.indexOf('pears');**

indexOf('dates')devuelve -1, indexOf('oranges')devuelve 2y indexOf('pears')devuelve 1(el primer índice en el que existe cada elemento).

**function quickCheck(arr, elem) {**

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

**return arr.indexOf(elem) != -1**

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

**}**

**console.log(quickCheck(['squash', 'onions', 'shallots'], 'mushrooms'));**

**Iterar a través de todos los elementos de una**

**matriz usando bucles for**

A veces, cuando se trabaja con matrices, es muy útil poder iterar a través de cada elemento para encontrar uno o más elementos que podamos necesitar, o manipular una matriz en función de qué elementos de datos cumplan con un determinado conjunto de criterios. JavaScript ofrece varios métodos integrados, cada uno de los cuales itera sobre las matrices de maneras ligeramente diferentes para lograr diferentes resultados (como every(), forEach(), map(), etc.), sin embargo, la técnica que es más flexible y nos ofrece la mayor cantidad de control es un forbucle simple.

Considera lo siguiente:

**function greaterThanTen(arr) {**

**let newArr = [];**

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

**if (arr[i] > 10) {**

**newArr.push(arr[i]);**

**}**

**}**

**return newArr;**

**}**

**greaterThanTen([2, 12, 8, 14, 80, 0, 1]);**

Usando un forbucle, esta función itera y accede a cada elemento de la matriz, y lo somete a una prueba simple que hemos creado. De esta forma, hemos determinado fácil y programáticamente qué elementos de datos son mayores que 10, y hemos devuelto una nueva matriz, [12, 14, 80]que contiene esos elementos.

**function filteredArray(arr, elem) {**

**let newArr = [];**

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

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

**if (arr[i].indexOf(elem) == -1) {**

**//Checks every parameter for the element and if is NOT there continues the code**

**newArr.push(arr[i]); //Inserts the element of the array in the new filtered array**

**}**

**}**

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

**return newArr;**

**}**

**console.log(filteredArray([[3, 2, 3], [1, 6, 3], [3, 13, 26], [19, 3, 9]], 3));**

**Cree matrices multidimensionales complejas**

¡Impresionante! ¡Acabas de aprender mucho sobre arreglos! Esta ha sido una descripción general de nivel bastante alto, y hay mucho más que aprender sobre cómo trabajar con arreglos, mucho de lo cual verá en secciones posteriores. Pero antes de pasar a mirar Objetos , echemos un vistazo más y veamos cómo las matrices pueden volverse un poco más complejas que lo que hemos visto en desafíos anteriores.

Una de las características más poderosas cuando se piensa en arreglos como estructuras de datos es que los arreglos pueden contener, o incluso estar completamente compuestos por otros arreglos. Hemos visto arreglos que contienen arreglos en desafíos anteriores, pero bastante simples. Sin embargo, los arreglos pueden contener una profundidad infinita de arreglos que pueden contener otros arreglos, cada uno con sus propios niveles arbitrarios de profundidad, y así sucesivamente. De esta manera, una matriz puede convertirse muy rápidamente en una estructura de datos muy compleja, conocida como matriz multidimensional o anidada. Considere el siguiente ejemplo:

**let nestedArray = [**

**['deep'],**

**[**

**['deeper'], ['deeper']**

**],**

**[**

**[**

**['deepest'], ['deepest']**

**],**

**[**

**[**

**['deepest-est?']**

**]**

**]**

**]**

**];**

La deepmatriz está anidada a 2 niveles de profundidad. Las deepermatrices tienen 3 niveles de profundidad. Las deepestmatrices son de 4 niveles y la deepest-est?de 5.

Si bien este ejemplo puede parecer intrincado, este nivel de complejidad no es desconocido, ni siquiera inusual, cuando se trata de grandes cantidades de datos. Sin embargo, todavía podemos acceder muy fácilmente a los niveles más profundos de una matriz tan compleja con notación de paréntesis:

**console.log(nestedArray[2][1][0][0][0]);**

Esto registra la cadena deepest-est?. Y ahora que sabemos dónde está ese dato, podemos restablecerlo si lo necesitamos:

**nestedArray[2][1][0][0][0] = 'deeper still';**

**console.log(nestedArray[2][1][0][0][0]);**

Ahora se registra deeper still.

**let myNestedArray = [**

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

**['unshift', false, 1, 2, 3, 'complex', 'nested'],**

**[['deep', 'loop', 'shift', 6, 7, 1000, 'method']],**

**[[['deeper', 'concat', false, true, 'spread', 'array']]],**

**[[[['deepest', 'mutate', 1327.98, 'splice', 'slice', 'push']]]],**

**['iterate', 1.3849, 7, '8.4876', 'arbitrary', 'depth']**

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

**];**

**Agregar pares de clave-valor a objetos de JavaScript**

En su forma más básica, los objetos son solo colecciones de pares clave-valor . En otras palabras, son piezas de datos ( valores ) asignados a identificadores únicos llamados propiedades ( claves ). Echa un vistazo a un ejemplo:

**const tekkenCharacter = {**

**player: 'Hwoarang',**

**fightingStyle: 'Tae Kwon Doe',**

**human: true**

**};**

El código anterior define un objeto de personaje de videojuego de Tekken llamado tekkenCharacter. Tiene tres propiedades, cada una de las cuales se asigna a un valor específico. Si desea agregar una propiedad adicional, como "origen", puede hacerlo asignando original objeto:

**tekkenCharacter.origin = 'South Korea';**

Esto usa la notación de puntos. Si tuviera que observar el tekkenCharacterobjeto, ahora incluirá la originpropiedad. Hwoarang también tenía un cabello anaranjado distintivo. Puede agregar esta propiedad con notación de corchetes haciendo:

**tekkenCharacter['hair color'] = 'dyed orange';**

Se requiere la notación entre paréntesis si su propiedad tiene un espacio o si desea usar una variable para nombrar la propiedad. En el caso anterior, la propiedad está entre comillas para indicarla como una cadena y se agregará exactamente como se muestra. Sin comillas, se evaluará como una variable y el nombre de la propiedad será el valor que tenga la variable. He aquí un ejemplo con una variable:

**const eyes = 'eye color';**

**tekkenCharacter[eyes] = 'brown';**

Después de agregar todos los ejemplos, el objeto se verá así:

**{**

**player: 'Hwoarang',**

**fightingStyle: 'Tae Kwon Doe',**

**human: true,**

**origin: 'South Korea',**

**'hair color': 'dyed orange',**

**'eye color': 'brown'**

**};**

**let foods = {**

**apples: 25,**

**oranges: 32,**

**plums: 28**

**};**

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

**foods["bananas"] = 13;**

**foods["grapes"] = 35;**

**foods["strawberries"] = 27;**

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

**console.log(foods);**

**let userActivity = {**

**id: 23894201352,**

**date: 'January 1, 2017',**

**data: {**

**totalUsers: 51,**

**online: 42**

**}**

**};**

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

**userActivity["data"]["online"] = 45;**

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

**console.log(userActivity);**

**Acceder a nombres de propiedades con notación de corchetes**

En el primer desafío de objetos, mencionamos el uso de la notación de corchetes como una forma de acceder a los valores de propiedad mediante la evaluación de una variable. Por ejemplo, imagine que nuestro foodsobjeto se está utilizando en un programa para una caja registradora de supermercado. Tenemos alguna función que establece el selectedFoody queremos comprobar nuestro foodsobjeto para la presencia de ese alimento. Esto podría verse como:

**let selectedFood = getCurrentFood(scannedItem);**

**let inventory = foods[selectedFood];**

Este código evaluará el valor almacenado en la selectedFoodvariable y devolverá el valor de esa clave en el foodsobjeto, o undefinedsi no está presente. La notación de corchetes es muy útil porque a veces las propiedades de los objetos no se conocen antes del tiempo de ejecución o necesitamos acceder a ellos de una manera más dinámica.

**let foods = {**

**apples: 25,**

**oranges: 32,**

**plums: 28,**

**bananas: 13,**

**grapes: 35,**

**strawberries: 27**

**};**

**function checkInventory(scannedItem) {**

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

**return foods[scannedItem]**

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

**}**

**console.log(checkInventory("apples"));**

**Use la palabra clave delete para eliminar las propiedades del objeto**

Ahora ya sabes qué son los objetos y sus características y ventajas básicas. En resumen, son almacenes de clave-valor que proporcionan una forma flexible e intuitiva de estructurar datos y proporcionan un tiempo de búsqueda muy rápido. A lo largo del resto de estos desafíos, describiremos varias operaciones comunes que puede realizar en objetos para que pueda sentirse cómodo aplicando estas estructuras de datos útiles en sus programas.

En desafíos anteriores, hemos agregado y modificado los pares clave-valor de un objeto. Aquí veremos cómo podemos eliminar un par clave-valor de un objeto.

Revisemos nuestro foodsejemplo de objeto por última vez. Si quisiéramos eliminar la applesclave, podemos eliminarla usando la deletepalabra clave como esta:

**delete foods.apples;**

**let foods = {**

**apples: 25,**

**oranges: 32,**

**plums: 28,**

**bananas: 13,**

**grapes: 35,**

**strawberries: 27**

**};**

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

**delete foods.oranges;**

**delete foods.plums;**

**delete foods.strawberries;**

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

**console.log(foods);**

**let users = {**

**Alan: {**

**age: 27,**

**online: true**

**},**

**Jeff: {**

**age: 32,**

**online: true**

**},**

**Sarah: {**

**age: 48,**

**online: true**

**},**

**Ryan: {**

**age: 19,**

**online: true**

**}**

**};**

**Comprobar si un objeto tiene una propiedad**

Ahora podemos agregar, modificar y eliminar claves de objetos. Pero, ¿y si solo quisiéramos saber si un objeto tiene una propiedad específica? JavaScript nos proporciona dos formas diferentes de hacer esto. Uno usa el hasOwnProperty()método y el otro usa la inpalabra clave. Si tenemos un objeto userscon una propiedad de Alan, podemos verificar su presencia de cualquiera de las siguientes maneras:

**users.hasOwnProperty('Alan');**

**'Alan' in users;**

Ambos regresarían true.

**function isEveryoneHere(userObj) {**

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

**return userObj.hasOwnProperty("Alan") &&**

**userObj.hasOwnProperty("Jeff") &&**

**userObj.hasOwnProperty("Sarah") &&**

**userObj.hasOwnProperty("Ryan");**

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

**}**

**console.log(isEveryoneHere(users));**

**Iterar a través de las claves de un**

**objeto con una instrucción for...in**

A veces, es posible que necesite iterar a través de todas las claves dentro de un objeto. Esto requiere una sintaxis específica en JavaScript llamada declaración for...in . Para nuestro usersobjeto, esto podría verse así:

**for (let user in users) {**

**console.log(user);**

**}**

Esto registraría Alan, Jeffy Sarah- cada valor en su propia línea.

En esta declaración, definimos una variable usery, como puede ver, esta variable se restableció durante cada iteración a cada una de las claves del objeto a medida que la declaración recorría el objeto, lo que resultó en que el nombre de cada usuario se imprimiera en la consola.

NOTA: Los objetos no mantienen un orden en las claves almacenadas como lo hacen las matrices; por lo tanto, la posición de una clave en un objeto, o el orden relativo en el que aparece, es irrelevante al hacer referencia o acceder a esa clave.

**const users = {**

**Alan: {**

**online: false**

**},**

**Jeff: {**

**online: true**

**},**

**Sarah: {**

**online: false**

**}**

**}**

**function countOnline(usersObj) {**

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

**let result = 0;**

**for (let user in usersObj) {**

**if (usersObj[user].online === true) {**

**result++;**

**}**

**}**

**return result;**

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

**}**

**console.log(countOnline(users));**

**enere una matriz de todas las claves de objeto con Object.keys()**

**También podemos generar una matriz que contenga todas las claves almacenadas en un objeto con el Object.keys( ) método. Este método toma un objeto como argumento y devuelve una matriz de cadenas que representan cada propiedad del objeto. Nuevamente, no habrá un orden específico para las entradas en la matriz.**

**let users = {**

**Alan: {**

**age: 27,**

**online: false**

**},**

**Jeff: {**

**age: 32,**

**online: true**

**},**

**Sarah: {**

**age: 48,**

**online: false**

**},**

**Ryan: {**

**age: 19,**

**online: true**

**}**

**};**

**function getArrayOfUsers(obj) {**

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

**return Object.keys(obj);**

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

**}**

**console.log(getArrayOfUsers(users));**

**Modificar una matriz almacenada en un objeto**

Ahora ha visto todas las operaciones básicas para los objetos de JavaScript. Puede agregar, modificar y eliminar pares clave-valor, verificar si existen claves e iterar sobre todas las claves en un objeto. A medida que continúe aprendiendo JavaScript, verá aplicaciones de objetos aún más versátiles. Además, las lecciones de Estructuras de datos ubicadas en la sección Preparación para la entrevista de codificación del plan de estudios también cubren los objetos ES6 Map y Set , los cuales son similares a los objetos ordinarios pero brindan algunas características adicionales. Ahora que ha aprendido los conceptos básicos de matrices y objetos, ¡está completamente preparado para comenzar a abordar problemas más complejos usando JavaScript!

**let user = {**

**name: 'Kenneth',**

**age: 28,**

**data: {**

**username: 'kennethCodesAllDay',**

**joinDate: 'March 26, 2016',**

**organization: 'freeCodeCamp',**

**friends: [**

**'Sam',**

**'Kira',**

**'Tomo'**

**],**

**location: {**

**city: 'San Francisco',**

**state: 'CA',**

**country: 'USA'**

**}**

**}**

**};**

**function addFriend(userObj, friend) {**

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

**userObj.data.friends.push(friend);**

**return userObj.data.friends;**

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

**}**

**console.log(addFriend(user, 'Pete'));**