

Basic Data Structures

Access an Array's Contents Using Bracket Notation

The fundamental feature of any data structure is, of course, the ability to not only store data, but to be able to retrieve that data on command. So, now that we've learned how to create an array, let's begin to think about how we can access that array's information.

When we define a simple array as seen below, there are 3 items in it:

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

In an array, each array item has an *index*. This index doubles as the position of that item in the array, and how you reference it. However, it is important to note, that JavaScript arrays are *zero-indexed*, meaning that the first element of an array is actually at the **zeroth** position, not the first. In order to retrieve an element from an array we can enclose an index in brackets and append it to the end of an array, or more commonly, to a variable which references an array object. This is known as *bracket notation*. For example, if we want to retrieve the `a` from `ourArray` and assign it to a variable, we can do so with the following code:

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

Now `ourVariable` has the value of `a`.

In addition to accessing the value associated with an index, you can also set an index to a value using the same notation:

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

Using bracket notation, we have now reset the item at index 1 from the string `b`, to `not b anymore`. Now `ourArray` is `["a", "not b anymore", "c"]`.

In order to complete this challenge, set the 2nd position (index 1) of `myArray` to anything you want, besides the letter `b`.

Solution:

```
let myArray = ["a", "b", "c", "d"];  
// Only change code below this line  
myArray[1] = "z";  
// Only change code above this line  
console.log(myArray);
```

Basic Data Structures

Use an Array to Store a Collection of Data

The below is an example of the simplest implementation of an array data structure. This is known as a *one-dimensional array*, meaning it only has one level, or that it does not have any other arrays nested within it. Notice it contains *booleans*, *strings*, and *numbers*, among other valid JavaScript data types:

```
let simpleArray = ['one', 2, 'three', true, false, undefined,  
null];  
console.log(simpleArray.length);
```

The `console.log` call displays 7.

All arrays have a `length` property, which as shown above, can be very easily accessed with the syntax `Array.length`. A more complex implementation of an array can be seen below. This is known as a *multi-dimensional array*, or an array that contains other arrays. Notice that this array also contains JavaScript *objects*, which we will examine very closely in our next section, but for now, all you need to know is that arrays are also capable of storing complex objects.

```
let complexArray = [
```

```
[
  {
    one: 1,
    two: 2
  },
  {
    three: 3,
    four: 4
  }
],
[
  {
    a: "a",
    b: "b"
  },
  {
    c: "c",
    d: "d"
  }
]
];
```

We have defined a variable called `yourArray`. Complete the statement by assigning an array of at least 5 elements in length to the `yourArray` variable. Your array should contain at least one *string*, one *number*, and one *boolean*.

Solution:

```
let yourArray = [1, 2, 3, "four", true]; // Change this 1
line
```

Access an Array's Contents Using Bracket Notation

The fundamental feature of any data structure is, of course, the ability to not only store data, but to be able to retrieve that data on command. So, now that we've learned how to create an array, let's begin to think about how we can access that array's information.

When we define a simple array as seen below, there are 3 items in it:

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

In an array, each array item has an *index*. This index doubles as the position of that item in the array, and how you reference it. However, it is important to note, that JavaScript arrays are *zero-indexed*, meaning that the first element of an array is actually at the **zeroth** position, not the first. In order to retrieve an element from an array we can enclose an index in brackets and append it to the end of an array, or more commonly, to a variable which references an array object. This is known as *bracket notation*. For example, if we want to retrieve the `a` from `ourArray` and assign it to a variable, we can do so with the following code:

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

Now `ourVariable` has the value of `a`.

In addition to accessing the value associated with an index, you can also *set* an index to a value using the same notation:

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

Using bracket notation, we have now reset the item at index 1 from the string `b`, to `not b anymore`. Now `ourArray` is `["a", "not b anymore", "c"]`.

In order to complete this challenge, set the 2nd position (index 1) of `myArray` to anything you want, besides the

Solution:

```
let myArray = ["a", "b", "c", "d"];
// Only change code below this line
myArray[1] = "z";
// Only change code above this line
console.log(myArray);
```

Add Items to an Array with `push()` and `unshift()`

An array's length, like the data types it can contain, is not fixed. Arrays can be defined with a length of any number of elements, and elements can be added or removed over time; in other words, arrays are *mutable*. In this challenge, we will look at two methods with which we can programmatically modify an array: `Array.push()` and `Array.unshift()`.

Both methods take one or more elements as parameters and add those elements to the array the method is being called on; the `push()` method adds elements to the end of an array, and `unshift()` adds elements to the beginning. Consider the following:

```
let twentyThree = 'XXIII';
let romanNumerals = ['XXI', 'XXII'];
```

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

`romanNumerals` would have the value `['XIX', 'XX', 'XXI', 'XXII']`.

```
romanNumerals.push(twentyThree);
```

`romanNumerals` would have the value `['XIX', 'XX', 'XXI', 'XXII', 'XXIII']`.

Notice that we can also pass variables, which allows us even greater flexibility in dynamically modifying our array's data.

We have defined a function, `mixedNumbers`, which we are passing an array as an argument. Modify the function by using `push()` and `unshift()` to add `'I', 2,`

'three' to the beginning of the array and 7, 'VIII', 9 to the end so that the returned array contains representations of the numbers 1-9 in order.

Solution:

```
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']));
```

Remove Items from an Array with `pop()` and `shift()`

Both `push()` and `unshift()` have corresponding methods that are nearly functional opposites: `pop()` and `shift()`. As you may have guessed by now, instead of adding, `pop()` *removes* an element from the end of an array, while `shift()` removes an element from the beginning. The key difference between `pop()` and `shift()` and their cousins `push()` and `unshift()`, is that neither method takes parameters, and each only allows an array to be modified by a single element at a time.

Let's take a look:

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

```
greetings.pop();
```

`greetings` would have the value `['whats up?', 'hello']`.

```
greetings.shift();
```

`greetings` would have the value `['hello']`.

We can also return the value of the removed element with either method like this:

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

`greetings` would have the value `[]`, and `popped` would have the value `hello`.

We have defined a function, `popShift`, which takes an array as an argument and returns a new array. Modify the function, using `pop()` and `shift()`, to remove the first and last elements of the argument array, and assign the removed elements to their corresponding variables, so that the returned array contains their values.

Solution:

```
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  
']));
```

Remove Items Using `splice()`

Ok, so we've learned how to remove elements from the beginning and end of arrays using `shift()` and `pop()`, but what if we want to remove an element from somewhere in the middle? Or remove more than one element at once? Well, that's where `splice()` comes in. `splice()` allows us to do just that: **remove any number of consecutive elements** from anywhere in an array.

`splice()` can take up to 3 parameters, but for now, we'll focus on just the first 2. The first two parameters of `splice()` are integers which represent indexes, or positions, of items in the array that `splice()` is being called upon. And remember, arrays are *zero-indexed*, so to indicate the first element of an array,

we would use `0`. `splice()`'s first parameter represents the index on the array from which to begin removing elements, while the second parameter indicates the number of elements to delete. For example:

```
let array = ['today', 'was', 'not', 'so', 'great'];  
  
array.splice(2, 2);
```

Here we remove 2 elements, beginning with the third element (at index 2). `array` would have the value `['today', 'was', 'great']`.

`splice()` not only modifies the array it's being called on, but it also returns a new array containing the value of the removed elements:

```
let array = ['I', 'am', 'feeling', 'really', 'happy'];  
  
let newArray = array.splice(3, 2);  
newArray has the value ['really', 'happy'].
```

We've initialized an array `arr`. Use `splice()` to remove elements from `arr`, so that it only contains elements that sum to the value of `10`.

Solution:

```
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);
```

Add Items Using `splice()`

Remember in the last challenge we mentioned that `splice()` can take up to three parameters? Well, you can use the third parameter, comprised of one or more element(s), to add to the array. This can be incredibly useful for quickly switching out an element, or a set of elements, for another.


```
const numbers = [10, 11, 12, 12, 15];
const startIndex = 3;
const amountToDelete = 1;

numbers.splice(startIndex, amountToDelete, 13, 14);
console.log(numbers);
```

The second occurrence of 12 is removed, and we add 13 and 14 at the same index. The `numbers` array would now be `[10, 11, 12, 13, 14, 15]`.

Here, we begin with an array of numbers. Then, we pass the following to `splice()`: The index at which to begin deleting elements (3), the number of elements to be deleted (1), and the remaining arguments (13, 14) will be inserted starting at that same index. Note that there can be any number of elements (separated by commas) following `amountToDelete`, each of which gets inserted.

We have defined a function, `htmlColorNames`, which takes an array of HTML colors as an argument. Modify the function using `splice()` to remove the first two elements of the array and add `'DarkSalmon'` and `'BlanchedAlmond'` in their respective places.

Solution:

```
function htmlColorNames(arr) {
  // Only change code below this line
  arr.splice(0, 2, "DarkSalmon", "BlanchedAlmond");
  // Only change code above this line
  return arr;
}

console.log(htmlColorNames(['DarkGoldenRod', 'WhiteSmoke',
  'LavenderBlush', 'PaleTurquoise', 'FireBrick']));
```

Copy Array Items Using `slice()`

The next method we will cover is `slice()`. Rather than modifying an array, `slice()` copies or *extracts* a given number of elements to a new array, **leaving the array it is called upon untouched**. `slice()` takes only 2 parameters — the first is the index at which to begin extraction, and the second is the index at which to stop extraction (extraction will occur up to, but not including the element at this index). Consider this:

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

```
let today'sWeather = weatherConditions.slice(1, 3);
```

`today'sWeather` would have the value `['snow', 'sleet']`, while `weatherConditions` would still have `['rain', 'snow', 'sleet', 'hail', 'clear']`.

In effect, we have created a new array by extracting elements from an existing array.

We have defined a function, `forecast`, that takes an array as an argument. Modify the function using `slice()` to extract information from the argument array and return a new array that contains the string elements `warm` and `sunny`.

Solution:

```
function forecast(arr) {  
  // Only change code below this line  
  let newArr = arr.slice(2, 4);  
  return newArr;  
}  
  
// Only change code above this line  
console.log(forecast(['cold', 'rainy', 'warm', 'sunny', 'cool', 'thunderstorms']));
```

Copy an Array with the Spread Operator

While `slice()` allows us to be selective about what elements of an array to copy, among several other useful tasks, ES6's new *spread operator* allows us to easily copy *all* of an array's elements, in order, with a simple and highly readable syntax. The spread syntax simply looks like this: ...

In practice, we can use the spread operator to copy an array like so:

```
let thisArray = [true, true, undefined, false, null];
let thatArray = [...thisArray];
```

`thatArray` equals `[true, true, undefined, false, null]`. `thisArray` remains unchanged and `thatArray` contains the same elements as `thisArray`.

We have defined a function, `copyMachine` which takes `arr` (an array) and `num` (a number) as arguments. The function is supposed to return a new array made up of `num` copies of `arr`. We have done most of the work for you, but it doesn't work quite right yet. Modify the function using spread syntax so that it works correctly (hint: another method we have already covered might come in handy here!).

Solution:

```
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], 3));
```

Combine Arrays with the Spread Operator

Another huge advantage of the *spread* operator is the ability to combine arrays, or to insert all the elements of one array into another, at any index. With more traditional syntaxes, we can concatenate arrays, but this only allows us to combine arrays at the end of one, and at the start of another. Spread syntax makes the following operation extremely simple:

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

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

thatArray would have the value ['basil', 'cilantro', 'sage', 'rosemary', 'parsley', 'thyme', 'coriander'].

Using spread syntax, we have just achieved an operation that would have been more complex and more verbose had we used traditional methods.

We have defined a function `spreadOut` that returns the variable `sentence`. Modify the function using the *spread* operator so that it returns the array ['learning', 'to', 'code', 'is', 'fun'].

Solution:

```
function spreadOut() {  
  let fragment = ['to', 'code'];  
  let sentence = ["learning", ...fragment, "is", "fun"];  
  // Change this line  
  return sentence;  
}
```

```
console.log(spreadOut());
```

Check For The Presence of an Element With `indexOf()`

Since arrays can be changed, or *mutated*, at any time, there's no guarantee about where a particular piece of data will be on a given array, or if that element even still exists. Luckily, JavaScript provides us with another built-in method, `indexOf()`, that allows us to quickly and easily check for the presence of an element on an array. `indexOf()` takes an element as a parameter, and when called, it returns the position, or index, of that element, or `-1` if the element does not exist on the array.

For example:

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

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

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

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

`indexOf('dates')` returns `-1`, `indexOf('oranges')` returns `2`,
and `indexOf('pears')` returns `1` (the first index at which each element exists).

`indexOf()` can be incredibly useful for quickly checking for the presence of an element on an array. We have defined a function, `quickCheck`, that takes an array and an element as arguments. Modify the function using `indexOf()` so that it returns `true` if the passed element exists on the array, and `false` if it does not.

Solution:

```
function quickCheck(arr, elem) {  
  // Only change code below this line  
  if (arr.indexOf(elem) == -1) {
```

```
    return false;
  } else {
    return true;
  }
  // Only change code above this line
}

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

Iterate Through All an Array's Items Using For Loops

Sometimes when working with arrays, it is very handy to be able to iterate through each item to find one or more elements that we might need, or to manipulate an array based on which data items meet a certain set of criteria. JavaScript offers several built in methods that each iterate over arrays in slightly different ways to achieve different results (such as `every()`, `forEach()`, `map()`, etc.), however the technique which is most flexible and offers us the greatest amount of control is a simple `for` loop.

Consider the following:

```
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]);
```

Using a `for` loop, this function iterates through and accesses each element of the array, and subjects it to a simple test that we have created. In this way, we have easily and programmatically determined which data items are greater than 10, and returned a new array, `[12, 14, 80]`, containing those items.

We have defined a function, `filteredArray`, which takes `arr`, a nested array, and `elem` as arguments, and returns a new array. `elem` represents an element that may or may not be present on one or more of the arrays nested within `arr`. Modify the function, using a `for` loop, to return a filtered version of the passed array such that any array nested within `arr` containing `elem` has been removed.

Solution:

```
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) {
      newArr.push(arr[i]);
    }
  }

  // Only change code above this line
  return newArr;
}

console.log(filteredArray([[3, 2, 3], [1, 6, 5], [3, 13, 26], [19, 4, 9]], 3));
```

Create complex multi-dimensional arrays

Awesome! You have just learned a ton about arrays! This has been a fairly high level overview, and there is plenty more to learn about working with arrays, much of which you will see in later sections. But before moving on to looking at *Objects*, let's take one more look, and see how arrays can become a bit more complex than what we have seen in previous challenges.

One of the most powerful features when thinking of arrays as data structures, is that arrays can contain, or even be completely made up of other arrays. We have seen arrays that contain arrays in previous challenges, but fairly simple ones. However, arrays can contain an infinite depth of arrays that can contain other arrays, each with their own arbitrary levels of depth, and so on. In this way, an array can very quickly become a very complex data structure, known as a *multi-dimensional*, or nested array. Consider the following example:

```
let nestedArray = [  
  ['deep'],  
  [  
    ['deeper'], ['deeper']  
  ],  
  [  
    [  
      ['deepest'], ['deepest']  
    ],  
    [  
      [  
        ['deepest-est?']  
      ]  
    ]  
  ]  
];
```

The `deep` array is nested 2 levels deep. The `deeper` arrays are 3 levels deep. The `deepest` arrays are 4 levels, and the `deepest-est?` is 5.

While this example may seem convoluted, this level of complexity is not unheard of, or even unusual, when dealing with large amounts of data. However, we can still very easily access the deepest levels of an array this complex with bracket notation:

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

This logs the string `deepest-est?`. And now that we know where that piece of data is, we can reset it if we need to:

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

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

Now it logs `deeper still`.

We have defined a variable, `myNestedArray`, set equal to an array. Modify `myNestedArray`, using any combination of *strings*, *numbers*, and *booleans* for data elements, so that it has exactly five levels of depth (remember, the outer-most array is level 1). Somewhere on the third level, include the string `deep`, on the fourth level, include the string `deeper`, and on the fifth level, include the string `deepest`.

Solution:

```
let myNestedArray = [
  // Only change code below this line
  ['unshift', false, 1, 2, 3, 'complex', 'nested'],
  ['loop', 'shift', 6, 7, 1000, 'method'],
  ['concat', false, true, 'spread', 'array', ["deep"]],
  ['mutate', 1327.98, 'splice', 'slice', 'push', [{"deeper"}]],
  ['iterate', 1.3849, 7, '8.4876', 'arbitrary', 'depth',
  [{"deepest"}]]
  // Only change code above this line
];
```

Add Key-Value Pairs to JavaScript Objects

At their most basic, objects are just collections of *key-value* pairs. In other words, they are pieces of data (*values*) mapped to unique identifiers called *properties* (*keys*). Take a look at an example:

```
const tekkenCharacter = {  
  player: 'Hwoarang',  
  fightingStyle: 'Tae Kwon Doe',  
  human: true  
};
```

The above code defines a Tekken video game character object called `tekkenCharacter`. It has three properties, each of which map to a specific value. If you want to add an additional property, such as "origin", it can be done by assigning `origin` to the object:

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

This uses dot notation. If you were to observe the `tekkenCharacter` object, it will now include the `origin` property. Hwoarang also had distinct orange hair. You can add this property with bracket notation by doing:

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

Bracket notation is required if your property has a space in it or if you want to use a variable to name the property. In the above case, the property is enclosed in quotes to denote it as a string and will be added exactly as shown. Without quotes, it will be evaluated as a variable and the name of the property will be whatever value the variable is. Here's an example with a variable:

```
const eyes = 'eye color';  
  
tekkenCharacter[eyes] = 'brown';
```

After adding all the examples, the object will look like this:

```
{
  player: 'Hwoarang',
  fightingStyle: 'Tae Kwon Doe',
  human: true,
  origin: 'South Korea',
  'hair color': 'dyed orange',
  'eye color': 'brown'
};
```

A `foods` object has been created with three entries. Using the syntax of your choice, add three more entries to it: `bananas` with a value of 13, `grapes` with a value of 35, and `strawberries` with a value of 27.

Solution:

```
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);
```

Modify an Object Nested Within an Object

Now let's take a look at a slightly more complex object. Object properties can be nested to an arbitrary depth, and their values can be any type of data supported by JavaScript, including arrays and even other objects. Consider the following:

```
let nestedObject = {  
  id: 28802695164,  
  date: 'December 31, 2016',  
  data: {  
    totalUsers: 99,  
    online: 80,  
    onlineStatus: {  
      active: 67,  
      away: 13,  
      busy: 8  
    }  
  }  
};
```

`nestedObject` has three properties: `id` (value is a number), `date` (value is a string), and `data` (value is an object with its nested structure). While structures can quickly become complex, we can still use the same notations to access the information we need. To assign the value `10` to the `busy` property of the nested `onlineStatus` object, we use dot notation to reference the property:

```
nestedObject.data.onlineStatus.busy = 10;
```

Here we've defined an object `userActivity`, which includes another object nested within it. Set the value of the `online` key to `45`.

Solution:

```
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);
```

Access Property Names with Bracket Notation

In the first object challenge we mentioned the use of bracket notation as a way to access property values using the evaluation of a variable. For instance, imagine that our `foods` object is being used in a program for a supermarket cash register. We have some function that sets the `selectedFood` and we want to check our `foods` object for the presence of that food. This might look like:

```
let selectedFood = getCurrentFood(scannedItem);
let inventory = foods[selectedFood];
```

This code will evaluate the value stored in the `selectedFood` variable and return the value of that key in the `foods` object, or `undefined` if it is not present. Bracket notation is very useful because sometimes object properties are not known before runtime or we need to access them in a more dynamic way.

We've defined a function, `checkInventory`, which receives a scanned item as an argument. Return the current value of the `scannedItem` key in the `foods` object.

You can assume that only valid keys will be provided as an argument to `checkInventory`.

Solution:

```
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 the delete Keyword to Remove Object Properties

Now you know what objects are and their basic features and advantages. In short, they are key-value stores which provide a flexible, intuitive way to structure data, **and**, they provide very fast lookup time. Throughout the rest of these challenges, we will describe several common operations you can perform on objects so you can become comfortable applying these useful data structures in your programs.

In earlier challenges, we have both added to and modified an object's key-value pairs. Here we will see how we can *remove* a key-value pair from an object.

Let's revisit our `foods` object example one last time. If we wanted to remove the `apples` key, we can remove it by using the `delete` keyword like this:

```
delete foods.apples;
```

Use the `delete` keyword to remove the `oranges`, `plums`, and `strawberries` keys from the `foods` object.

Solution:

```
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);
```

Check if an Object has a Property

Now we can add, modify, and remove keys from objects. But what if we just wanted to know if an object has a specific property? JavaScript provides us with two different ways to do this. One uses the `hasOwnProperty()` method and

the other uses the `in` keyword. If we have an object `users` with a property of `Alan`, we could check for its presence in either of the following ways:

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

```
'Alan' in users;
```

Both of these would return `true`.

Finish writing the function so that it returns `true` if the object passed to it contains all four names, `Alan`, `Jeff`, `Sarah` and `Ryan` and returns `false` otherwise.

Solution:

```
let users = {
  Alan: {
    age: 27,
    online: true
  },
  Jeff: {
    age: 32,
    online: true
  },
  Sarah: {
    age: 48,
    online: true
  },
  Ryan: {
    age: 19,
    online: true
  }
};
```

```
function isEveryoneHere(userObj) {
  // Only change code below this line
  if (
    "Alan" in userObj &&
```



```
    "Jeff" in userObj &&  
    "Sarah" in userObj &&  
    "Ryan" in userObj  
  ) {  
    return true;  
  } else {  
    return false;  
  }  
  // Only change code above this line  
}  
  
console.log(isEveryoneHere(users));
```

Iterate Through the Keys of an Object with a `for...in` Statement

Sometimes you may need to iterate through all the keys within an object. This requires a specific syntax in JavaScript called a *for...in* statement. For our `users` object, this could look like:

```
for (let user in users) {  
  console.log(user);  
}
```

This would log `Alan`, `Jeff`, and `Sarah` - each value on its own line.

In this statement, we defined a variable `user`, and as you can see, this variable was reset during each iteration to each of the object's keys as the statement looped through the object, resulting in each user's name being printed to the console.

NOTE: Objects do not maintain an ordering to stored keys like arrays do; thus a key's position on an object, or the relative order in which it appears, is irrelevant when referencing or accessing that key.

We've defined a function `countOnline` which accepts one argument (a users object). Use a *for...in* statement within this function to loop through the users object passed into the function and return the number of users whose `online` property is set to `true`. An example of a users object which could be passed to `countOnline` is shown below. Each user will have an `online` property with either a `true` or `false` value.

```
{
  Alan: {
    online: false
  },
  Jeff: {
    online: true
  },
  Sarah: {
    online: false
  }
}
```

Solution:

```
const users = {
  Alan: {
    online: false
  },
  Jeff: {
    online: true
  },
  Sarah: {
    online: true
  }
}
```

```
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));
```

Generate an Array of All Object Keys with Object.keys()

We can also generate an array which contains all the keys stored in an object with the `Object.keys()` method. This method takes an object as the argument and returns an array of strings representing each property in the object. Again, there will be no specific order to the entries in the array.

Finish writing the `getArrayOfUsers` function so that it returns an array containing all the properties in the object it receives as an argument.

Solution:

```
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));
```

Modify an Array Stored in an Object

Now you've seen all the basic operations for JavaScript objects. You can add, modify, and remove key-value pairs, check if keys exist, and iterate over all the keys in an object. As you continue learning JavaScript you will see even more versatile applications of objects. Additionally, the Data Structures lessons located in the Coding Interview Prep section of the curriculum also cover the ES6 *Map* and *Set* objects, both of which are similar to ordinary objects but provide some additional features. Now that you've learned the basics of arrays and objects, you're fully prepared to begin tackling more complex problems using JavaScript!

Take a look at the object we've provided in the code editor. The `user` object contains three keys. The `data` key contains five keys, one of which contains an array of `friends`. From this, you can see how flexible objects are as data structures. We've started writing a function `addFriend`. Finish writing it so that

it takes a `user` object and adds the name of the `friend` argument to the array stored in `user.data.friends` and returns that array.

Solution:

```
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'));
```