

**Coner Murphy**

Mar 24 · 39 tweets · 12 min read



Bookmark

Save as PDF

+ My Authors

📢 Massive JavaScript Resource Collection 📢

All 37 JavaScript Array methods (current & experimental) explained in images 📌

1 Array .at()

.at() lets us access elements in an Array by providing an index.

Both positive and negative ones are supported. 📌

JS

JavaScript

Array.at()

A positive index access the array at the start and counts forward.

```
1 const array1 = [5, 12, 8, 130, 44];
2
3 let index = 2;
4
5 console.log(`Using an index of ${index} the item returned is ${array1.at(index)}`);
6 // expected output: "Using an index of 2 the item returned is 8"
7
8 index = -2;
9
10 console.log(`Using an index of ${index} item returned is ${array1.at(index)}`);
11 // expected output: "Using an index of -2 item returned is 130"
```

A negative index access the array at the end and counts backwards.

1 Array.at(index)

Array.at() takes one parameter:


- index – the index you want to access the array at.

@MrConerMurphy

2 Array .concat()

This method lets us join multiple arrays into one while leaving the original one untouched.

New arrays are added to the end of the previous ones. 📌



JavaScript

Array.concat()

Call the `.concat()` method on the array we wish to add onto.

```

1 const array1 = [1,2,3,4,5];
2 const array2 = [6,7,8,9,0];
3
4 const array3 = array1.concat(array2);
5
6 console.log(array1); // [1,2,3,4,5]
7 console.log(array2); // [6,7,8,9,0]
8 console.log(array3); // [1,2,3,4,5,6,7,8,9,0]

```

A new Array is created and the originals are left unmodified.


```

1 concat()
2 concat(value0)
3 concat(value0, value1)
4 concat(value0, value1, ..., valueN)

```

`concat()` takes 1 parameter:


- valueN** – Arrays and/or values to concatenate into a new array.

 @MrConerMurphy

3 Array.copyWithin()

`.copyWithin()` let's copy elements within an array to another location within that array. 📌

Array.copyWithin()

 @MrConerMurphy

The original array is modified by using `.copyWithin()`.

```

1 const array = [0,1,2,3,4];
2
3 const newArray = array.copyWithin(0, 2);
4
5 console.log(newArray); // [2,3,4,3,4]

```

```

1 const array = [0,1,2,3,4];
2
3 const newArray = array.copyWithin(0, 2, 4);
4
5 console.log(newArray); // [2,3,2,3,4]
6 console.log(array); // [2,3,2,3,4]

```

```

1 const array = [0,1,2,3,4];
2
3 const newArray = array.copyWithin(-3,-2,-1);
4
5 console.log(newArray); // [0,1,3,3,4]

```

Negative indexes start counting from the end of the array.

```

1 const array = [0,1,2,3,4];
2
3 const newArray = array.copyWithin(2);
4
5 console.log(newArray); // [0,1,0,1,2]

```

`copyWithin()` takes 3 parameters:

- target** – index in array to copy the values to.
- start*** – index to start copying elements from.
- end*** – index to end copying elements, not included in the copy.

*optional parameter

```

1 copyWithin(target)
2 copyWithin(target, start)
3 copyWithin(target, start, end)

```

4 Array.entries()

`.entries()` returns a key/value pair for each index in the array, we can then loop through the iterator using a `for...of` loop. 📌

Array.entries()



@MrConerMurphy

```
1 const carMakers = ["Ford", "Fiat", "Ferrari"];
2
3 const carIterator = carMakers.entries();
4
5 console.log(carIterator.next().value);
6 // [0, "Ford"]
```

`.entries()` returns an Array iterator containing key/value pairs for each index in the array

```
1 const carMakers = ["Ford", "Fiat", "Ferrari"];
2
3 const carIterator = carMakers.entries();
4
5 for (let [index, element] of carIterator) {
6   console.log(element);
7   // Ford
8   // Fiat
9   // Ferrari
10 }
```

Use destructuring to get just the element in the iterator

```
1 const carMakers = ["Ford", "Fiat", "Ferrari"];
2
3 const carIterator = carMakers.entries();
4
5 for (let maker of carIterator) {
6   console.log(maker);
7   // [0, "Ford"]
8   // [1, "Fiat"]
9   // [2, "Ferrari"]
10 }
```

Use a for-of loop to iterator over the iterator to access the values.

`.entries()` takes 0 parameters, instead we call it on the Array we want to make the iterator from.

```
1 Array.entries()
```

5 Array.every()

`.every()` lets us test every value in an array against a provided function, if they all pass it returns true, otherwise false. 📌

Array.every()



@MrConerMurphy

We test all values against a function we pass to `.every()`.

```
1 const evenTester = (num) => num % 2 === 0;
2
3 const list1 = [1, 2, 6, 7, 10];
4 const list2 = [2, 4, 6, 8, 10];
5
6 console.log(list1.every(evenTester)); // false
7 console.log(list2.every(evenTester)); // true
```

If all items pass the test in the function, then true is returned. If one or more fails, false.

`.every()` takes 4 parameters:

- callbackFn (takes 3 params):
 - element: current item being processed
 - index: index of the current item
 - array: the array `.every()` was called on
- thisArg: a value to use this within the callbackFn

```
1 every(function(element, index, array) { /* ... */ }, thisArg)
```

6 Array.fill()

`.fill()` lets us fill either an entire array or part of one with a given value. 📌

Array.fill()



@MrConerMurphy



```
1 const array = [1,2,3];
2
3 array.fill(0);
4
5 console.log(array); // [0, 0, 0]
```

Pass the value we want to fill the array with. By default it will fill the entire array.

.fill() is a mutator method, so the array it's called upon will be changed.

.fill() takes 3 parameters:

- **value**: value to fill the array with.
- **start***: Start index (inclusive), 0 if not given.
- **end***: End index (exclusive), array.length if not given.

*Optional Parameter



```
1 fill(value, start, end)
```

Using the optional start and end parameters to fill a selection of the array.



```
1 const array = [1,2,3,4,5];
2
3 array.fill(0,2,4);
4
5 console.log(array); // [1, 2, 0, 0, 5]
```

7 Array.filter()

.filter() allows us to filter an array to only the values that pass a provided test function. 📌

Array.filter()



@MrConerMurphy

Values in the array .filter() is called upon is tested against the test in the callbackFn provided.



```
1 const numbers = [1, 2, 3, 4, 5, 6];
2
3 const evens = numbers.filter((num) => num % 2 === 0);
4 const odds = numbers.filter((num) => num % 2 !== 0);
5 const zero = numbers.filter((num) => num % 0 === 0);
6
7 console.log(evens); // [2,4,6]
8 console.log(odds); // [1,3,5]
9 console.log(zero); // []
```

.filter() takes 2 parameters:

- **callbackFn** (takes 3 arguments):
 - **element**: current item being processed
 - **index***: index of the current item
 - **array***: the array .filter() was called on
- **thisArg***: a value to use this within the callbackFn

*Optional



```
1 filter(function(element, index, array) { /* ... */ }, thisArg)
```

Values that pass the provided test are returned into a new array, leaving the original as before.

If no values pass the test, then an empty array is returned.

8 Array.find()

.find() allow us to test each item in an array, the first value that passes the test is returned. 📌

Array.find()



@MrConerMurphy

Pass a function to test each item of the array. `.find` is called on against. The first value to pass the test is returned.



```
1 const array1 = [5, 12, 8, 130, 44];
2 const array2 = [5, 2, 8, 6, 4];
3
4 const found1 = array1.find(element => element > 10);
5 const found2 = array2.find(element => element > 10);
6
7 console.log(found1); // expected output: 12
8 console.log(found2); // expected output: undefined
```

If no values pass the provided test then `undefined` is returned.

`.find()` takes 2 parameters:

- `callbackFn` (takes 3 arguments):
 - `element`: current item being processed
 - `index*`: index of the current item
 - `array*`: the array `.find()` was called on
- `thisArg*`: a value to use this within the `callbackFn`

*Optional



```
1 find(function(element, index, array) { /* ... */ }, thisArg)
```

9 Array.findIndex()

Similar to `.find()` but this time instead of returning the first element that passes the test, we return the index of it.



Array.findIndex()



@MrConerMurphy

`.findIndex()` returns the index of the first item that passes the testing function provided to it.

If no item passes `-1` is returned.



```
1 const array = [5, 9, 8, 130, 44];
2
3 console.log(array.findIndex((item) => item > 10));
4 // expected output: 3
```

`.findIndex()` takes 2 parameters:

- `callbackFn` (takes 3 arguments):
 - `element`: current item being processed
 - `index*`: index of the current item
 - `array*`: the array `.findIndex()` was called on
- `thisArg*`: a value to use this within the `callbackFn`

*Optional



```
1 findIndex(function(element, index, array) { /* ... */ }, thisArg)
```

1 0 Array.flat()

`.flat()` lets us flatten nested arrays down levels. We can control the number of levels flattened by using the `depth` parameter. 🖱️

Array.flat()



@MrConerMurphy

By passing a depth parameter to `.flat()` we flatten nested arrays up to that depth.

```
1 const arr1 = [0, 1, 2, [3, 4]];
2 console.log(arr1.flat()); // [0,1,2,3,4]
3
4 const arr2 = [0, 1, 2, [[3, 4]]];
5 console.log(arr2.flat(2)); // [0,1,2,3,4]
6
7 const arr3 = [0, 1, 2, [[[3, 4]]]];
8 console.log(arr3.flat(Infinity)); // [0,1,2,3,4]
```

We can pass `Infinity` to `.flat()` to flatten an array regardless of how many nested arrays there are.

`.flat()` takes 1 parameter:

- **depth***: the level a nested array should be flattened to. Defaults to 1

*Optional

```
1 flat()
2 flat(depth)
```

1 1 Array.flatMap()

`.flatMap()` is like calling a `.map()` followed by a `.flat()` but it will only let us flatten one level of arrays. 📌



@MrConerMurphy

Array.flatMap()

`.flatMap()` is identical to calling `.map()` followed by `.flat()` but is more efficient.

`.flatMap()` takes 2 parameters:

- **callbackFn** (takes 3 arguments):
 - **currentValue**: current item being processed
 - **index***: index of the current item
 - **array***: the array `.flatMap()` was called on
- **thisArg***: a value to use this within the callbackFn

*Optional

```
1 flatMap(function(currentValue, index, array) { /* ... */ }, thisArg)
```


```
1 const array = ['String 1', 'String 2', 'String 3']
2
3 array.map(str => str.split(' '));
4 // [['String','1'], ['String','2'], ['String','3']]
5
6 array.flatMap(str => str.split(' '));
7 // ['String','1', 'String','2', 'String','3']
```

`.flatMap()` will only flatten 1 level of arrays and this can't be changed. So, if more than 1 is needed, use `.map()` and `.flat()`

1 2 Array.forEach()

`.forEach()` allows us to iterate over an array. `.forEach()` does not return any values from the array, only undefined.



 @MrConerMurphy

Array.forEach()

.forEach() is a alternative to using traditional loops for iterating over arrays.

.forEach() takes 2 parameters:

- **callbackFn** (takes 3 arguments):
 - **element**: current item being processed
 - **index***: index of element
 - **array***: the array **.forEach()** was called on
- **thisArg***: a value to use this within the callbackFn

*Optional


```
1 forEach(function(element, index, array) { /* ... */ }, thisArg)
```

```
1 const array = [1,2,3,4,5];
2
3 array.forEach((num) => {
4   console.log(num * 2)
5 })
```

.forEach() returns undefined. You cannot return anything from **.forEach()**, if you want to manipulate the array into a new one, use **.map()** or **.reduce()**. Also, you cannot chain **.forEach()**.

1 3 Array.from()

.from() lets us create a new array from an array-like value, we can also map over each item as we create the new array 🙋

 @MrConerMurphy

Array.from()

We call **.from()** from Array directly rather than on the value we want to apply it to like is common with Array methods.

.from() takes 3 parameters:

- **arrayLike**: An array-like or iterable object to convert to an array.
- **mapFn***: A function to call on every element in the array.
- **thisArg***: A value to use this within the callbackFn

*Optional


```
1 Array.from(arrayLike, function mapFn(element, index) { /* ... */ }, thisArg)
```

```
1 const string = "string";
2
3 const array = Array.from(string);
4 const mappedArray = Array.from(string, (letter) => letter + " ");
5
6 console.log(array); // ["s", "t", "r", "i", "n", "g"]
7 console.log(mappedArray); // ["s ", "t ", "r ", "i ", "n ", "g "]
```

We can pass a function to map over each item in the Array as we create it. This saves us having to chain **.map()** onto **.from()**.

1 4 Array.includes()

.includes() lets us search an array for a provided value if it's found in the array, true is returned otherwise false 🙋

 @MrConerMurphy

Array.includes()

.includes() let's us search an Array for the provided value, if found it returns true, otherwise false.

.includes() takes 2 parameters:

- **searchElement**: The value to search for in the Array. (case-sensitive).
- **fromIndex***: The index to start searching the Array from.

*Optional

```
1 includes(searchElement, fromIndex)
```


```
1 const array = [0, 1, 2, 3, 4, 5];
2
3 const includes3 = array.includes(3);
4 const includes3FromIndex5 = array.includes(3, 5);
5 const includes4 = array.includes(4, -1);
6
7 console.log(includes3); // true
8 console.log(includes3FromIndex5); // false
9 console.log(includes4); // false
```

We can use negative indexes to start searching from the end of the Array. In this case it returns false because we're only searching the last element (5).

1 5 Array.indexOf()

.indexOf() lets us search an array for a given value, if it's found it's index is returned.

.indexOf() will return on the first occurrence of the value 📍

 @MrConerMurphy

Array.indexOf()

.indexOf() let's us search an Array for a provided value, if found it returns the index of the first occurrence of the item.

.indexOf() takes 2 parameters:

- **searchElement**: The value to search for in the Array. (case-sensitive).
- **fromIndex***: The index to start searching the Array from.

*Optional


```
1 indexOf(searchElement, fromIndex)
```

```
1 const numbers = [2, 3, 4, 6, 2];
2
3 console.log(numbers.indexOf(6, 2)); // 3
4 console.log(numbers.indexOf(2)); // 0
5 console.log(numbers.indexOf(10)); // -1
```

If the *searchElement* provided is not found in the array then -1 is returned.

1 6 Array.isArray()

.isArray() allows us to test if a provided value is an Array or not. 📍

 @MrConerMurphy

Array.isArray()

Any Array passed to `.isArray()` will return true, regardless if it is empty or not, using `new Array()` or `[]`.

`.isArray()` takes 1 parameters:

- value:** The value to be checked if is an Array or not.


```
Array.isArray([1, 2, 3]); // true
Array.isArray(new Array()); // true
Array.isArray({foo: 123}); // false
Array.isArray('foobar'); // false
Array.isArray(undefined); // false
```

Any value that is not an Array will return false including if nothing is passed.

```
Array.isArray(value)
```

1 7 Array .join()

`.join()` takes a provided string and joins each element in the array with it. 🖱️

 @MrConerMurphy

Array.join()

With no separator parameter passed, `.join()` will join the values with a `,`.

`.join()` takes 1 parameter:

- separator*:** the string to join elements of the array with.

*Optional

```
1 const arr = ["hello", "world", "JS", "is", "amazing", [], undefined, null];
2
3 console.log(arr.join()); // hello,world,JS,is,amazing,,,
4 console.log(arr.join("-")); // hello-world-JS-is-amazing---
5 console.log(arr.join("")); // helloworldJSisamazing
6 console.log(arr.join(" ")); // hello world JS is amazing
7
```


Multiple spaces and tabs are preserved in `.join()` so can create longer gaps.

Null, undefined and `[]` are converted to empty strings.

```
1 join(separator)
```

1 8 Array .keys()

`.keys()` creates an iterator that contains all of the indexes in the array, black spaces aren't ignored. 🖱️

 @MrConerMurphy

Array.keys()

```
1 var arr = ["a", , "c"];
2 var sparseKeys = Object.keys(arr);
3 var denseKeys = [...arr.keys()];
4 console.log(sparseKeys); // ['0', '2']
5 console.log(denseKeys); // [0, 1, 2]
```

```
1 const array1 = ['a', 'b', 'c'];
2 const iterator = array1.keys();
3
4 for (const key of iterator) {
5   console.log(key);
6 }
7
8 // expected output: 0
9 // expected output: 1
10 // expected output: 2
```

.keys() takes 0 parameters


```
1 keys()
```

.keys() doesn't ignore blank spaces in Arrays. All values in the Array are included regardless if they are empty or not.

.keys() creates a iterator that contains the keys for each index in the array.

1 9 Array.lastIndexOf()

.lastIndexOf() returns the last index that a provided element is found at in an array. 📍

 @MrConerMurphy

Array.lastIndexOf()

.lastIndexOf() takes 2 parameters:

- searchElement:** The value to search for in the Array. (case-sensitive).
- fromIndex*:** The index to start searching backwards from in the Array.

*Optional

```
1 lastIndexOf(searchElement, fromIndex)
2
```

If the array length + the fromIndex is less than 0 the array isn't searched, -1 is returned.

The index of the last occurrence of the searchElement is returned, -1 if not found.


```
1 const numbers = [1, 20, 40, 3, 56];
2
3 console.log(numbers.lastIndexOf(20)); // 1
4 console.log(numbers.lastIndexOf(40, -2)); // 2
5 console.log(numbers.lastIndexOf(100)); // -1
6 console.log(numbers.lastIndexOf(1, -7)); // -1
7 console.log(numbers.lastIndexOf(56, 100)); // 4
8
```

Because we are searching backwards, if we specify a fromIndex larger than the array length it still works.

2 0 Array.groupBy()

.groupBy() is one of the newest array methods, it allows us to group objects in an array by properties they share.



 @MrConerMurphy

Array.groupBy()

.groupBy() takes 2 parameters:

- callbackFn** (takes 3 arguments):
 - element**: current item being processed
 - index***: index of element
 - array***: the array **.groupBy()** was called on
- thisArg***: a value to use this within the callbackFn

*Optional

```
1 groupBy(function(element, index, array) { /* ... */ }, thisArg)
2
```

snappify.io

```
1 const items = [
2   {name: 'bananas', type: 'fruit'},
3   {name: 'goat', type: 'meat'},
4   {name: 'cherries', type: 'fruit'}
5 ]
6
7 const grouped = items.groupBy(({type}) => type);
8
9 console.log(grouped);
10
11 /*
12 {
13   fruit: [
14     { name: "bananas", type: "fruit" },
15     { name: "cherries", type: "fruit" }
16   ],
17   meat: [
18     { name: "goat", type: "meat" }
19   ]
20 }
21 */
```


snappify.io

Group an array of objects by a property shared by the objects.

! **.groupBy()** is only supported in Firefox Nightly at present.

2 1 Array.groupByToMap()

.groupByToMap() was added along with **.groupBy()**, it allows us to group objects like **.groupBy()** but it then allows us to add them to a Map object based on a property. 📌

 @MrConerMurphy

Array.groupByToMap()

.groupByToMap() lets us group elements in an array by using a testing function.

.groupByToMap() takes 2 parameters:

- callbackFn** (takes 3 arguments):
 - element**: current item being processed
 - index***: index of element
 - array***: the array it was called on
- thisArg***: a value to use this within the callbackFn

*Optional

```
1 groupByToMap(function(element, index, array) { /* ... */ }, thisArg)
```

snappify.io


```
1 const children = {};
2 const adults = {};
3
4 const people = [{name: 'Bob', age: 20}, {name: 'Jess', age: 6}, {name: 'Alfie', age: 3}, {name: 'Chelsea', age: 20}];
5
6 const results = people.groupByToMap(({age}) => age < 18 ? children : adults);
7
8 results.get(adults); // [{name: 'Bob', age: 20}, {name: 'Chelsea', age: 20}]
9 results.get(children); // [{name: 'Jess', age: 6}, {name: 'Alfie', age: 3}]
```

We need to use **.get()** to retrieve the final array as the returned value from **.groupByToMap()** is a Map.

! **.groupByToMap()** is only supported in Firefox Nightly at present.

2 2 Array.map()

.map() lets us iterate over an array and return a new array with the new values from any manipulations we perform. 📌

 @MrConerMurphy

Array.map()

If you don't need to return an array then `.forEach()` is a better method to use (if you don't need a return value at all) or use a `for...of` loop.

`.map()` takes 2 parameters:

- **callbackFn** (takes 3 arguments):
 - **element**: current item being processed
 - **index***: index of element
 - **array***: the array `.map()` was called on
- **thisArg***: a value to use this within the callbackFn

*Optional


```
1 map(function(element, index, array) { /* ... */ }, thisArg)
2
```

```
1 const numbers = [1, 2, 3, 4, 5, 6];
2
3 const squares = numbers.map((num) => num * num);
4
5 console.log(squares); // [1, 4, 9, 16, 25, 36]
6
```

`.map()` creates a new array which contains the results of calling a function on every element in the calling array.

2 3 Array.of()

`.of()` allows us to create a new array from the values we pass to it, regardless of the quantity or their type. 📌

 @MrConerMurphy

Array.of()

`Array.of()` creates a new Array from the arguments passed to it, regardless of quantity or type of them.

```
1 console.log(Array.of(2, "hello", true));
2 // [2, "hello", true]
3 console.log(Array(2)); // [undefined, undefined]
4
```

`.of()` takes 1 parameter:

- **elementN**: The elements used to create the array.


```
1 Array.of(element0, element1, /* ... */ elementN)
2
```

The difference between `Array.of()` and the `Array` constructor is the handling of integers.

`Array.of()` will create an Array containing the number, where the `Array` constructor will create an Array of empty slots of that length.

2 4 Array.pop()

`.pop()` removes the last item from the array and returns it. 📌

 @MrConerMurphy

Array.pop()

.pop() removes the last element from the array it's called upon and returns it to the caller.

.pop() returns the removed element from the array or undefined if the array is empty.

.pop() takes 0 parameters:

```
1 pop()
```

snappify.io


```
1 const numbers = [1, 2, 3, 4];
2
3 console.log(numbers.pop()); // 4
4
5 console.log(numbers); // [1,2,3]
6
```

snappify.io

.pop() changes the original array and its length.

2 5 Array.push()

.push() allows us to add multiple new items to an array, it returns the updated array's length. 📌

 @MrConerMurphy

Array.push()

.push() returns the new length of the array after the items passed have been added to the array.

.push() takes 1 parameter:

- elementN:** The elements to add to the end of the array

```
1 push(element0, element1, /* ... */ elementN)
2
```

snappify.io


```
1 const shop = ["Apple", "Chicken", "Grapes"];
2
3 const newItem = shop.push("Oranges");
4 console.log(newItem); // 4
5
6 shop.push("Beef", "Blackcurrants");
7
8 console.log(shop);
9 // 📌 ["Apple", "Chicken", "Grapes", "Oranges", "Beef", "Blackcurrants"]
10
```

snappify.io

.push() let's you add multiple items to the end of the array it was called upon.

2 6 Array.reduce()

.reduce() lets us define a custom reducer function to be performed on each element in the array, going from left to right in the array. 📌

 @MrConerMurphy

Array.reduce()

.reduce() goes through the array element by element and performs the defined action with the previous and current values.

.reduce() takes 2 parameters:

- callbackFn** (takes 4 arguments):
 - previousValue**: the value of the previous call to callbackFn
 - currentValue**: the value of the current element.
 - currentIndex**: the index position of currentValue in the array.
 - array**: The array **.reduce()** was called upon
- initialValue***: value to which previousValue is initialized as.

*Optional

Here we define a **initialValue** of 1 to start the reduce with.

By not giving an **initialValue**, **array[0]** is taken as it and **currentValue** is **array[1]**.

```


1 const numbers = [1, 2, 3, 4, 5, 6];
2
3 const sum = numbers.reduce((pre, cur) => {
4   return pre + cur;
5 }, 1); // 22
6
7 const product = numbers.reduce((pre, cur) => {
8   return pre * cur;
9 }); // 720
  
```

```

1 reduce(function(previousValue, currentValue, currentIndex, array) { /* ... */ }, initialValue)
2
  
```

2 7 Array.reduceRight()

.reduceRight() is the same as **.reduce()** but instead going from right to left in the array. 📌

 @MrConerMurphy

Array.reduceRight()

.reduceRight() works the same as **.reduce()** by reducing an array down to a single value.

.reduceRight() goes right-to-left when processing the array, **.reduce()** goes left-to-right.

.reduceRight() takes 2 parameters:

- callbackFn** (takes 4 arguments):
 - accumulator**: the value of the previous call to callbackFn
 - currentValue**: the value of the current element.
 - index**: the index position of currentValue in the array.
 - array**: The array **.reduceRight()** was called upon
- initialValue***: value to which previousValue is initialized as.

*Optional

```


1 const array = [[0, 1],[2, 3],[4, 5]];
2
3 const newArray = array.reduceRight((acc, cur) => acc.concat(cur));
4
5 console.log(newArray);
6 // 📌 [4, 5, 2, 3, 0, 1]
  
```

```

1 reduceRight(function(accumulator, currentValue, index, array) { /* ... */ }, initialValue)
2
  
```

2 8 Array.reverse()

.reverse() will reverse an array in place, meaning the first item becomes the last and the last the first. 📌

 @MrConerMurphy

Array.reverse()

`.reverse()` will reverse an array in place meaning the first element becomes the last and the last element, the first.

`.reverse()` takes 0 parameters:


```
1 reverse()
```

```
1 const arr = [0, 1, 2, 3, 4, 5];
2
3 const reversed = arr.reverse();
4 console.log(reversed); // [5, 4, 3, 2, 1, 0]
5
6 console.log(arr); // [5, 4, 3, 2, 1, 0]
```

`.reverse()` is destructive so the original array is also changed.

2 9 Array .shift()

`.shift()` removes the first item from the array and returns it. 🖱️

 @MrConerMurphy

Array.shift()

`.shift()` will modify the original array it is called upon by removing the first item. It also modifies the length of the array.

`.shift()` takes 0 parameters:


```
1 shift()
```

```
1 const arr = [0, 1, 2, 3, 4, 5];
2
3 const item = arr.shift();
4
5 console.log(arr); // [1, 2, 3, 4, 5]
6 console.log(item); // 0
```

`.shift()` removes the first item from the array and returns it.

3 0 Array .slice()

`.slice()` allows you to take a copy of an array either as a whole or part of it. The original array is left untouched. 🖱️

 @MrConerMurphy

Array.slice()

.slice() takes a shallow copy of the array (or portion of it). The original array is not modified.

.slice() takes 2 parameters:

- **start***: Index in the array to start extraction of values from.
- **end***: Index to stop extraction at (excluded from it).

*Optional

```
1 slice(start, end)
```

```


1 const arr = [0, 1, 2, 3, 4, 5, 6];
2
3 const slice1 = arr.slice(3);
4 console.log(slice1); // [3, 4, 5, 6]
5 console.log(arr); // [0, 1, 2, 3, 4, 5, 6]
6
7 const slice2 = arr.slice(2, 5);
8 console.log(slice2); // [2, 3, 4]
9
10 const slice3 = arr.slice();
11 console.log(slice3); // [0, 1, 2, 3, 4, 5, 6]

```

If no start or end values are passed, the array is copied as a whole.

3 1 Array .some()

.some() checks if at least one item in an array passes the provided test function, if so it returns true otherwise false. 📌

 @MrConerMurphy

Array.some()

.some() takes 2 parameters:

- **callbackFn** (takes 3 arguments):
 - **element**: current item being processed
 - **index***: index of element
 - **array***: the array **.some()** was called on
- **thisArg***: a value to use this within the callbackFn

*Optional

```
1 some(function(element, index, array) { /* ... */ }, thisArg)
```

```


1 const arr = [1, 2, 3, 4, 5, 6];
2
3 const divisibleBy3 = arr.some((num) => num % 3 === 0);
4 const divisibleBy7 = arr.some((num) => num % 7 === 0);
5
6 console.log(divisibleBy3); // true
7 console.log(divisibleBy7); // false
8

```

.some() checks if at least one item in the array passes the test function provided. If one passes, it returns true, otherwise false.

3 2 Array .sort()

.sort() allows you to sort an array based on a provided sorting function. 📌

 @MrConerMurphy

Array.sort()

`.sort()`'s `compareFn` takes 2 arguments to compare. The order they are compared in, controls asc/dec order in the final array.

`.sort()` sorts the array in place, meaning the original array is modified to be the sorted one.

`.sort()` takes 1 parameter:

- `compareFn*` (takes 2 arguments):
 - `a`: first element to compare
 - `b`: second element to compare

*Optional


```
1 sort(function compareFn(a, b) { /* ... */ })
```

```
1 const arr = [10, 40, 2, 34, 31, 19];
2
3 const sortedDec = [...arr].sort((a, b) => b - a);
4 console.log(sortedDec); // [40, 34, 31, 19, 10, 2]
5
6 const sortedAsc = arr.sort((a, b) => a - b);
7 console.log(sortedAsc); // [2, 10, 19, 31, 34, 40]
8 console.log(arr); // [2, 10, 19, 31, 34, 40]
9
10 const sortNoComparison = [...arr].sort();
11 console.log(sortNoComparison); // [10, 19, 2, 31, 34, 40]
```

If no `compareFn` is provided to `.sort()` then the values are converted to strings and sorted according to each characters Unicode point value.

3 3 Array .splice()

`.splice()` allows you to remove or replace items in an array and return the removed/replaced items. 🖱️

 @MrConerMurphy

Array.splice()

`.splice()` returns the values removed/replaced from the original array.

`.splice()` takes 3 parameters:

- `start`: index to start changing the array
- `deleteCount*`: number of items to remove
- `item1, ...*`: items to add to the array starting at `start`

*Optional


```
1 splice(start, deleteCount, item1, item2, itemN)
```

```
1 const arr = [15, 23, 1, 3, 13, 20];
2
3 arr.splice(5, 1, "foo");
4
5 arr.splice(2, 2, "hello", "world");
6
7 const spliced = arr.splice(4, 1);
8 console.log(spliced); // [13]
9
10 console.log(arr); // [15, 23, "hello", "world", "foo"]
```

`.splice()` changes the original array, so if you want the original array to remain unchanged, take a copy of the array and use `.splice()` on that.

3 4 Array .toLocaleString()

`.toLocaleString()` will convert the array it's called on to a string and then separate the string with a locale-specific string (e.g. ',') 🖱️

 @MrConerMurphy

Array.toLocaleString()

Each item in the original array is converted to a string, the strings are then separated by a locale-specific string (e.g. ",")

.toLocaleString() takes 2 parameters:

- **locales***: A string/array of strings containing a language tag(s).
- **options***: An object containing the configuration properties to apply to the array items.

*Optional


```
1 toLocaleString(locales, options);
```

```
1 const items = [100, "Water", 56];
2
3 const pounds = items.toLocaleString("en-GB", {
4   style: "currency",
5   currency: "GBP"
6 });
7
8 const dollars = items.toLocaleString("en-US", {
9   style: "currency",
10  currency: "USD"
11 });
12
13 console.log(pounds); // £100.00,Water,£56.00
14 console.log(dollars); // $100.00,Water,$56.00
```

We can format number and date types using their own **.toLocaleString()** methods and the options available on it.

3 5 Array.toString()

.toString() will recursively flatten an array until it's one level and convert it to a string separated by a comma. 📌

 @MrConerMurphy

Array.toString()

Arrays are flattened recursively until all levels are flattened.

```
1 const items = [100, "Water", true, ["Hello", ["World", "!"]]];
2
3 const string = items.toString();
4
5 console.log(string); // 100,Water,true,Hello,World,!
```


.toString() takes 0 parameters.

```
1 toString()
```

When **.toString()** is called on an array, all the items are joined together and separated by a "," in the returned string.

3 6 Array.unshift()

.unshift() adds one or more values provided to the beginning of the array and returns the updated length of the array. 📌

 @MrConerMurphy

Array.unshift()

.unshift() returns the length of the array after the new values have been added.

.unshift() takes 1 parameter:

- elementN: The values to add to the start of the array

.unshift() adds one or more values to the beginning of the array that it is called on.

```


1 const array = [0, 1, 2, 3, 4];
2
3 const value = array.unshift(5, 6, 7);
4
5 console.log(value); // 8
6 console.log(array); // [5, 6, 7, 0, 1, 2, 3, 4]
  
```

```

1 unshift(element0, element1, /* ... */ elementN)
  
```

3 7 Array.values()

.values() returns an iterator object containing a value for every index in the array which we can then loop over with a **for...of** loop. 🙌

 @MrConerMurphy

Array.values()

.values() returns a new array iterator object that contains the values for every index in the array.

We can loop through the values using a **for...of** loop. Alternatively, you can use **.next()** to access the values like we do with generators.

.values() takes 0 parameters.

```

1 const array = [0, 1, 2];
2
3 const iterator = array.values();
4
5 console.log(iterator);
6
7 for (const value of iterator) {
8   console.log(value);
9 }
10
11 // Output: 0
12 // Output: 1
13 // Output: 2
  
```

```

1 values()
  
```

Did you enjoy this thread/find it helpful?

If so please consider doing one of the below to show your support:

❤ Like these tweets

🔄 Retweet the 1st tweet in this thread so others can see it.

👤 Follow me ([@MrConerMurphy](#)) for more content like this.

🐦 Follow Us on Twitter!

🐦 Tweet

📄 Share