# Fun with Native Arrays

*In JavaScript, arrays can be created with the `Array` constructor, or using the `[]` convenience shortcut, which is also the preferred approach. Arrays inherit from the `Object` prototype and they haven't a special value for `typeof`, they return `'object'` too. Using `[] instanceof Array`, however, returns true. That being said, there are also Array-like objects which complicate matters, such as strings, or the `arguments` object. The `arguments` object is not an instance of `Array`, but it still has a `length` property, and its values are indexed, so it can be looped like any Array.*

Nicolás Bevacqua

*Published 3 years ago | 22 minute read | 4*

In this article I'll go over a few of the methods on the `Array` prototype, and explore the possibilities each of these methods unveil.

- Looping with `.forEach`
- Asserting with `.some` and `.every`
- Subtleties in `.join` and `.concat`
- Stacks and queues with `.pop`, `.push`, `.shift`, and `.unshift`
- Model mapping with `.map`
- Querying with `.filter`

- Ordering with `.sort`
- Computing with `.reduce`, `.reduceRight`
- Copying a `.slice`
- The power of `.splice`
- Lookups with `.indexOf`
- The `in` operator
- Going in `.reverse`

```
> Array
  function Array() { [native code] }
> []
  []
> typeof []
  "object"
> [] instanceof Array
  true
> (function(){ return arguments instanceof Array })()
  false
> ['_', 't', 'a', 'n', 'i', 'f', ']'].forEach(function (value, index, array) {
      this.push(String.fromCharCode(value.charCodeAt() + index + 2))
  }, out = [])

  out.join('')
  "awesome"
> |
```

*console.png*

You can copy and paste any of the examples in your browser's console, I sure did!

## Looping with `.forEach`

This is one of the simplest methods in a native JavaScript Array. Unsurprisingly unsupported™ in

IE7 and IE8.

`forEach` takes a callback which is invoked once for each element in the array, and gets passed three arguments.

- `value` containing the current array element
- `index` is the element's position in the array
- `array` is a reference to the array

Furthermore, we could pass an optional second argument which will become the context ( `this` ) for each function call.

```
['_', 't', 'a', 'n', 'i', 'f', ']'].forEach(function (value, index, array) {
  this.push(String.fromCharCode(value.charCodeAt() + index + 2))
}, out = [])

out.join('')
// <- 'awesome'
```

I cheated with `.join` which we didn't cover *yet*, but we'll look at it soon. In this case, it joins together the different elements in the array, effectively doing something like `out[0] + '' + out[1] + '' + out[2] + '' + out[n]` . We **can't break `forEach` loops**, and throwing exceptions wouldn't be very sensible. Luckily, we have other options available to us in those cases where we might want to short-circuit a loop.

## Asserting with `.some` and `.every`

If you've ever worked with .NET's enumerables, these methods are the *poorly named* cousins of `.Any(x => x.IsAwesome)` and `.All(x => x.IsAwesome)`.

These methods are similar to `.forEach` in that they also take a callback with `value`, `index`, and `array`, which can be context-bound passing a second argument. The MDN docs describe `.some`:

---

`some` executes the callback function once for each element present in the array until it finds one where `callback` returns a `true` value. If such an element is found, `some` immediately returns `true`. Otherwise, `some` returns `false`. `callback` is invoked only for indexes of the array which have assigned values; it is not invoked for indexes which have been deleted or which have never been assigned values.

---

```
max = -Infinity
satisfied = [10, 12, 10, 8, 5, 23].some(function (value, index, array) {
  if (value > max) max = value
  return value < 10
})

console.log(max)
// <- 12

satisfied
// <- true
```

Note that the function stopped looping after it hit the first item which satisfied the callback's condition `value < 10`. `.every` works in the same way, but short-circuits happen when your callback returns `false`, rather than `true`.

## Subtleties in `.join` and `.concat`

The `.join` method is often confused with `.concat`. `.join(separator)` creates a string, resulting of taking every element in the array and separating them by `separator`. If no `separator` is provided, it'll default to a comma `','`. `.concat` works by creating new arrays which are shallow copies of the source arrays.

- `.concat` has the signature: `array.concat(val, val2, val3, valn)`
- `.concat` returns a new array
- `array.concat()` with no arguments returns a shallow copy of the array

Shallow copy means that the copy will hold the same object references as the source array, which is generally a good thing. For example:

```
var a = { foo: 'bar' }
var b = [1, 2, 3, a]
var c = b.concat()

console.log(b === c)
// <- false

b[3] === a && c[3] === a
// <- true
```

## Stacks and queues with `.pop`, `.push`, `.shift`, and `.unshift`

Nowadays, everyone knows that *adding elements to the end of an array* is done using `.push`. Did you know that you can push many elements at once using `[].push('a', 'b', 'c', 'd', 'z')`?

The `.pop` method is the counterpart of the most common use for `.push`. It'll return the last element in the array, and remove it from the array at the same time. If the array is empty, `void 0` ( `undefined` ) is returned. Using `.push` and `.pop` we could easily create a LIFO (last in first out) stack.

```
function Stack () {
  this._stack = []
}

Stack.prototype.next = function () {
  return this._stack.pop()
}

Stack.prototype.add = function () {
  return this._stack.push.apply(this._stack, arguments)
}

stack = new Stack()
stack.add(1,2,3)

stack.next()
// <- 3
```

Inversely, we could create a FIFO (first in first out) queue using `.unshift` and `.shift`.

```
function Queue () {
  this._queue = []
}

Queue.prototype.next = function () {
  return this._queue.shift()
}

Queue.prototype.add = function () {
```

```
  return this._queue.unshift.apply(this._queue, arguments)
}

queue = new Queue()
queue.add(1,2,3)

queue.next()
// <- 1
```

Using `.shift` (or `.pop`) is an easy way to loop through a set of array elements, while draining the array in the process.

```
list = [1,2,3,4,5,6,7,8,9,10]

while (item = list.shift()) {
  console.log(item)
}

list
// <- []
```

## Model mapping with `.map`

`map` calls a provided callback function once for each element in an array, in order, and constructs a new array from the results. `callback` is invoked only for indexes of the array which have assigned values; it is not invoked for indexes which have been deleted or which have never been assigned values.

The `Array.prototype.map` method has the same signature we've seen in `.forEach`, `.some`, and `.every`: `.map(fn(value, index, array), thisArgument)`.

```
values = [void 0, null, false, '']
values[7] = void 0
result = values.map(function(value, index, array){
    console.log(value)
    return value
})

// <- [undefined, null, false, '', undefined × 3, undefined]
```

The `undefined × 3` values explain that while `.map` won't run for deleted or unassigned array elements, they'll be still included in the resulting array. Mapping is very useful for casting or transforming arrays.

```
// casting
[1, '2', '30', '9'].map(function (value) {
  return parseInt(value, 10)
})
// 1, 2, 30, 9

[97, 119, 101, 115, 111, 109, 101].map(String.fromCharCode).join('')
// <- 'awesome'

// a commonly used pattern is mapping to new objects
items.map(function (item) {
  return {
    id: item.id,
    name: computeName(item)
  }
})
```

## Querying with `.filter`

`filter` calls a provided callback function once for each element in an array, and constructs a

new array of all the values for which `callback` returns a `true` value. `callback` is invoked only for indexes of the array which have assigned values; it is not invoked for indexes which have been deleted or which have never been assigned values. Array elements which do not pass the `callback` test are simply skipped, and **are not included** in the new array.

---

Same as usual: `.filter(fn(value, index, array), thisArgument)`. Think of it as the `.Where(x => x.IsAwesome)` LINQ expression (if you're into C#), or the `WHERE` SQL clause. Considering `.filter` only returns elements which pass the `callback` test with a truthy value, there are some interesting use cases.

```
[void 0, null, false, '', 1].filter(function (value) {
  return value
})
// <- [1]

[void 0, null, false, '', 1].filter(function (value) {
  return !value
})
// <- [void 0, null, false, '']
```

## Ordering with `.sort(compareFunction)`

---

If `compareFunction` is not supplied, elements are sorted by converting them to strings and comparing strings in lexicographic ("dictionary" or "telephone book," not numerical) order. For example, "80" comes before "9" in lexicographic order, but in a numeric sort 9 comes before 80.

---

Like most sorting functions, `Array.prototype.sort(fn(a,b))` takes a callback which tests two elements, and should produce one of three return values:

- return value `< 0` if `a` comes before `b`
- return value `=== 0` if both `a` and `b` are considered equivalent
- return value `> 0` if `a` comes after `b`

```
[9,80,3,10,5,6].sort()
// <- [10, 3, 5, 6, 80, 9]

[9,80,3,10,5,6].sort(function (a, b) {
  return a - b
})
// <- [3, 5, 6, 9, 10, 80]
```

## Computing with `.reduce`, `.reduceRight`

Reduce functions are, at first, hard to wrap our heads around. These functions loop through the array, from left-to-right *( `.reduce` )* or right-to-left *( `.reduceRight` )*, each invocation receives the partial result so far, and the operation results in a single aggregated return value.

Both methods have the following signature: `.reduce(callback(previousValue, currentValue, index, array), initialValue)`.

The `previousValue` will be the value returned in the last callback invocation, or `initialValue` the first time around. `currentValue` contains the current element, while `index` indicates the array position for the element. `array` is simply a reference to the array `.reduce` was called on.

One of the typical use cases for `.reduce` is the sum function.

```
Array.prototype.sum = function () {
```

```
  return this.reduce(function (partial, value) {
    return partial + value
  }, 0)
};

[3,4,5,6,10].sum()
// <- 28
```

Say we wanted to join a few strings together. We could use `.join` to that purpose. In the case of objects, though, `.join` wouldn't work as we expected, unless the objects had a reasonable `valueOf` or `toString` representation. However, we might use `.reduce` as a string builder for those objects.

```
function concat (input) {
  return input.reduce(function (partial, value) {
    if (partial) {
      partial += ', '
    }
    return partial + value.name
  }, '')
}

concat([
  { name: 'George' },
  { name: 'Sam' },
  { name: 'Pear' }
])
// <- 'George, Sam, Pear'
```

## Copying a `.slice`

Similarly to `.concat`, calls to `.slice` without any arguments produce a shallow copy of the source array. Slice takes two arguments, a `begin` and an `end` position.

`Array.prototype.slice` can be used to convert array like objects into real arrays

`Array.prototype.slice` can be used to convert array-like objects into real arrays.

```
Array.prototype.slice.call({ 0: 'a', 1: 'b', length: 2 })
// <- ['a', 'b']
```

This won't work with `.concat`, because it'll wrap the array-like object in a real array, instead.

```
Array.prototype.concat.call({ 0: 'a', 1: 'b', length: 2 })
// <- [{ 0: 'a', 1: 'b', length: 2 }]
```

Other than that, another common use for `.slice` is *removing the first few elements* from a list of arguments (an array-like object, which we could cast to a real array).

```
function format (text, bold) {
  if (bold) {
    text = '<b>' + text + '</b>'
  }
  var values = Array.prototype.slice.call(arguments, 2)

  values.forEach(function (value) {
    text = text.replace('%s', value)
  })

  return text
}

format('some%sthing%s %s', true, 'some', 'other', 'things')
// <- <b>somesomethingother things</b>
```

## The power of `.splice`

`.splice` is one of my favorite native array functions. It allows you to remove elements, insert new ones, and to do both in the same position, using just one function call. Note that this function alters the source array, unlike `.concat` or `.slice`.

```
var source = [1,2,3,8,8,8,8,8,9,10,11,12,13]
var spliced = source.splice(3, 4, 4, 5, 6, 7)

console.log(source)
// <- [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 ,13]

spliced
// <- [8, 8, 8, 8]
```

As you might've noted, it also returns the removed elements. This might come in handy if you want to loop a section of the array and then forget about it.

```
var source = [1,2,3,8,8,8,8,8,9,10,11,12,13]
var spliced = source.splice(9)

spliced.forEach(function (value) {
  console.log('removed', value)
})
// <- removed 10
// <- removed 11
// <- removed 12
// <- removed 13

console.log(source)
// <- [1, 2, 3, 8, 8, 8, 8, 8, 9]
```

## Lookups with `.indexOf`

With `.indexOf`, we can look up array element positions. If it can't find a match, `-1` is returned.

A pattern I find myself using a lot, is when I have comparisons such as `a === 'a' || a === 'b' || a === 'c'`, or even with just two comparsions. You could just use `.indexOf`, like so: `['a', 'b', 'c'].indexOf(a) !== -1`.

Note that objects will be found only if the same reference is provided. A second argument can provide the start index at which to begin searching.

```
var a = { foo: 'bar' }
var b = [a, 2]

console.log(b.indexOf(1))
// <- -1

console.log(b.indexOf({ foo: 'bar' }))
// <- -1

console.log(b.indexOf(a))
// <- 0

console.log(b.indexOf(a, 1))
// <- -1

b.indexOf(2, 1)
// <- 1
```

If you want to go in the reverse direction, `.lastIndexOf` will do the trick.

## The `in` operator

A common rookie mistake during interviews is to confuse `.indexOf` with the `in` operator, and hand-scribbling things such as:

```
var a = [1, 2, 5]

1 in a
// <- true, but because of the 2!

5 in a
// <- false
```

The problem here was that the `in` operator checks the object key for a value, rather than searching for values. This is, of course, much faster than using `.indexOf`.

```
var a = [3, 7, 6]

1 in a === !!a[1]
// <- true
```

The `in` operator is similar to casting the value at the provided key to a boolean value. The `!!` expression is used by some developers to negate a value, and then negate it again. *Effectively casting to boolean* any truthy value to `true`, and any falsy value to `false`.

## Going in `.reverse`

This method will take the elements in an array and reverse them in place.

```
var a = [1, 1, 7, 8]

a.reverse()
// [8, 7, 1, 1]
```

Rather than a copy, the array itself is modified. In a future article we'll expand on these concepts to see how we could create an _ -like library, such as Underscore or Lo-Dash.