APRIL 11, 2019 / #JAVASCRIPT

How to clone an array in JavaScript

Yazeed Bzadough

JavaScript has many ways to do anything. I've written on 10 Ways to Write pipe/compose in JavaScript, and now we're doing arrays.

1. Spread Operator (Shallow copy)

Ever since ES6 dropped, this has been the most popular method. It's a brief syntax and you'll find it incredibly useful when using libraries like React and Redux.

```
numbers = [1, 2, 3];
numbersCopy = [...numbers];
```

Note: This doesn't safely copy multi-dimensional arrays. Array/object values are copied by *reference* instead of by *value*.

This is fine

```
numbersCopy.push(4);
console.log(numbers, numbersCopy);
// [1, 2, 3] and [1, 2, 3, 4]
// numbers is left alone
```

This is not fine

```
nestedNumbers = [[1], [2]];
numbersCopy = [...nestedNumbers];

numbersCopy[0].push(300);
console.log(nestedNumbers, numbersCopy);
// [[1, 300], [2]]
// [[1, 300], [2]]
// They've both been changed because they share references
```

2. Good Old for() Loop (Shallow copy)

I imagine this approach is the *least* popular, given how trendy functional programming's become in our circles.

Pure or impure, declarative or imperative, it gets the job done!

```
numbers = [1, 2, 3];
numbersCopy = [];

for (i = 0; i < numbers.length; i++) {
  numbersCopy[i] = numbers[i];
}
```

Note: This doesn't safely copy multi-dimensional arrays. Since you're using the `=` operator, it'll assign objects/arrays by *reference* instead of by *value*.

This is fine

```
numbersCopy.push(4);
console.log(numbers, numbersCopy);
// [1, 2, 3] and [1, 2, 3, 4]
// numbers is left alone
```

This is not fine

```
nestedNumbers = [[1], [2]];
numbersCopy = [];

for (i = 0; i < nestedNumbers.length; i++) {
  numbersCopy[i] = nestedNumbers[i];
}

numbersCopy[0].push(300);
console.log(nestedNumbers, numbersCopy);
// [[1, 300], [2]]
// [[1, 300], [2]]
// They've both been changed because they share references
```

3. Good Old while() Loop (Shallow copy)

Same as `for` —impure, imperative, blah, blah, blah…it works! ?

```
numbers = [1, 2, 3];
numbersCopy = [];
i = -1;

while (++i < numbers.length) {
  numbersCopy[i] = numbers[i];
}
```

Note: This also assigns objects/arrays by *reference* instead of by *value*.

This is fine

```
numbersCopy.push(4);
console.log(numbers, numbersCopy);
// [1, 2, 3] and [1, 2, 3, 4]
// numbers is left alone
```

This is not fine

```
nestedNumbers = [[1], [2]];
numbersCopy = [];

i = -1;

while (++i < nestedNumbers.length) {
  numbersCopy[i] = nestedNumbers[i];
}

numbersCopy[0].push(300);
console.log(nestedNumbers, numbersCopy);
```

```
// [[1, 300], [2]]
// [[1, 300], [2]]
// They've both been changed because they share references
```

4. Array.map (Shallow copy)

Back in modern territory, we'll find the `map` function. Rooted in mathematics, `map` is the concept of transforming a set into another type of set, while preserving structure.

In English, that means `Array.map` returns an array of the same length every single time.

To double a list of numbers, use `map` with a `double` function.

```
numbers = [1, 2, 3];
double = (x) => x * 2;

numbers.map(double);
```

What about cloning??

True, this article's about cloning arrays. To duplicate an array, just return the element in your `map` call.

```
numbers = [1, 2, 3];
numbersCopy = numbers.map((x) => x);
```

If you'd like to be a bit more mathematical, `(x) => x` is called *identity*. It returns whatever parameter it's been given.

`map(identity)` clones a list.

```
identity = (x) => x;
numbers.map(identity);
// [1, 2, 3]
```

Note: This also assigns objects/arrays by *reference* instead of by *value*.

Donate

5. Array.filter (Shallow copy)

This function returns an array, just like `map`, but it's not guaranteed to be the same length.

What if you're filtering for even numbers?

```
[1, 2, 3].filter((x) => x % 2 === 0);
// [2]
```

The input array length was 3, but the resulting length is 1.

If your `filter`'s predicate always returns `true`, however, you get a duplicate!

```
numbers = [1, 2, 3];
numbersCopy = numbers.filter(() => true);
```

Every element passes the test, so it gets returned.

Note: This also assigns objects/arrays by *reference* instead of by *value*.

6. Array.reduce (Shallow copy)

I almost feel bad using `reduce` to clone an array, because it's so much more powerful than that. But here we go…

```
numbers = [1, 2, 3];

numbersCopy = numbers.reduce((newArray, element) => {
  newArray.push(element);

  return newArray;
}, []);
```

`reduce` transforms an initial value as it loops through a list.

Here the initial value is an empty array, and we're filling it with each element as we go. Donate
That array must be returned from the function to be used in the next iteration.

Note: This also assigns objects/arrays by *reference* instead of by *value*.

7. Array.slice (Shallow copy)

`slice` returns a *shallow* copy of an array based on the provided start/end index you provide.

If we want the first 3 elements:

```
[1, 2, 3, 4, 5].slice(0, 3);
// [1, 2, 3]
// Starts at index 0, stops at index 3
```

If we want **all** the elements, don't give any parameters

```
numbers = [1, 2, 3, 4, 5];
numbersCopy = numbers.slice();
// [1, 2, 3, 4, 5]
```

Note: This is a *shallow* copy, so it also assigns objects/arrays by *reference* instead of by *value*.

8. JSON.parse and JSON.stringify (Deep copy)

`JSON.stringify` turns an object into a string.

`JSON.parse` turns a string into an object.

Combining them can turn an object into a string, and then reverse the process to create a brand new data structure.

Note: This one safely copies deeply nested objects/arrays!

```
nestedNumbers = [[1], [2]];
numbersCopy = JSON.parse(JSON.stringify(nestedNumbers));

numbersCopy[0].push(300);
console.log(nestedNumbers, numbersCopy);

// [[1], [2]]
// [[1, 300], [2]]
// These two arrays are completely separate!
```

9. Array.concat (Shallow copy)

`concat` combines arrays with values or other arrays.

```
[1, 2, 3].concat(4); // [1, 2, 3, 4]
[1, 2, 3].concat([4, 5]); // [1, 2, 3, 4, 5]
```

If you give nothing or an empty array, a shallow copy's returned.

```
[1, 2, 3].concat(); // [1, 2, 3]
[1, 2, 3].concat([]); // [1, 2, 3]
```

Note: This also assigns objects/arrays by *reference* instead of by *value*.

10. Array.from (Shallow copy)

This can turn any iterable object into an array. Giving an array returns a shallow copy.

```
numbers = [1, 2, 3];
numbersCopy = Array.from(numbers);
// [1, 2, 3]
```

Note: This also assigns objects/arrays by *reference* instead of by *value*.

Conclusion

Well, this was fun ?

I tried to clone using just 1 step. You'll find many more ways if you employ multiple methods and techniques.

Donate

Yazeed Bzadough

Front-End Developer creating content at https://yazeedb.com.

If you read this far, tweet to the author to show them you care. Tweet a thanks

Learn to code for free. freeCodeCamp's open source curriculum has helped more than 40,000 people get jobs as developers. Get started

ADVERTISEMENT

Trending Guides

Best Python IDEs

Python Index