How to Deep Copy Objects and Arrays in JavaScript

The usual methods of copying an object or array only make a shallow copy, so deeply-nested references are a problem. You need a deep copy if a JavaScript object contains other objects.

Dr. Derek Austin Follow
Oct 7, 2019 · 8 min read ★

Photo by João Silas on Unsplash

What is a shallow copy?

Making a shallow copy of an array or object means creating new references to the primitive values inside the object, copying them.

That means that changes to the original array will not affect the copied array, which is what would happen if only the reference to the array had been copied (such as would occur with the assignment operator `=`).

A shallow copy refers to the fact that only one level is copied, and that will work fine for an array or object containing only primitive values.

For objects and arrays containing other objects or arrays, copying these objects requires a deep copy. Otherwise, changes made to the nested references will change the data nested in the original object or array.

In this article, I describe 4 methods of making a shallow copy and then 5 methods of making a deep copy in JavaScript.

Photo by Jakob Owens on Unsplash

Shallow copy using ...

1. The spread operator (...) is a convenient way to make a shallow copy of an array or object —when there is no nesting, it works great.

```
const array = ['😉','😋','😇']

const copyWithEquals = array // Changes to array will change copyWithEquals
console.log(copyWithEquals === array) // true (The assignment operator did not make a copy)

const copyWithSpread = [...array] // Changes to array will not change copyWithSpread
console.log(copyWithSpread === array) // false (The spread operator made a shallow copy)

array[0] = '😠' // Whoops, a bug

console.log(...array) // 😠 😋 😇
console.log(...copyWithEquals) // 😠 😋 😇
console.log(...copyWithSpread) // 😉 😋 😇
```

Copying arrays with the spread operator.js hosted with ♡ by GitHub view raw

As shown above, the spread operator is useful for creating new instances of arrays that do not behave unexpectedly due to old references. The spread operator is thus useful for adding to an array in React State.

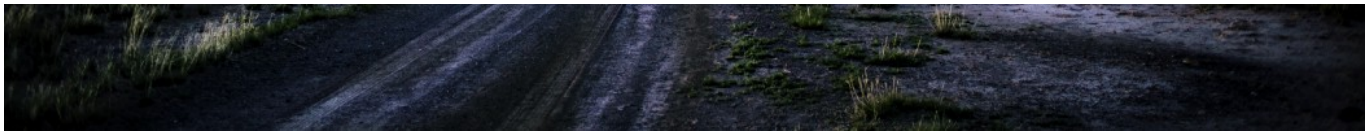

Photo by Donald Giannatti on Unsplash

Shallow copy using `.slice()`

2. For arrays specifically, using the built-in `.slice()` method works the same as the spread operator — creating a shallow copy of one level:

```
const array = ['😉','😋','😇']

const copyWithEquals = array // Changes to array will change copyWithEquals
console.log(copyWithEquals === array) // true (The assignment operator did not make a copy)

const copyWithSlice = array.slice() // Changes to array will not change copyWithSlice
console.log(copyWithSlice === array) // false (Using .slice() made a shallow copy of the array)

array[0] = '😠' // Whoops, a bug

console.log(...array) // 😠 😋 😇
console.log(...copyWithEquals) // 😠 😋 😇
console.log(...copyWithSlice) // 😉 😋 😇
```

Copying arrays with .slice().is hosted with ♡ by GitHub view raw

Photo by Antonio Garcia on Unsplash

Shallow copy using .assign()

3. The same type of shallow copy would be created using `Object.assign()`, which can be used with any object or array:

```
const array = ['😉','😋','😇']

const copyWithEquals = array // Changes to array will change copyWithEquals
const copyWithAssign = [] // Changes to array will not change copyWithAssign
Object.assign(copyWithAssign, array) // Object.assign(target, source)

array[0] = '😠' // Whoops, a bug

console.log(...array) // 😠 😋 😇
console.log(...copyWithEquals) // 😠 😋 😇
console.log(...copyWithAssign) // 😉 😋 😇
```

Copying arrays with Object.assign().js hosted with ♡ by GitHub view raw

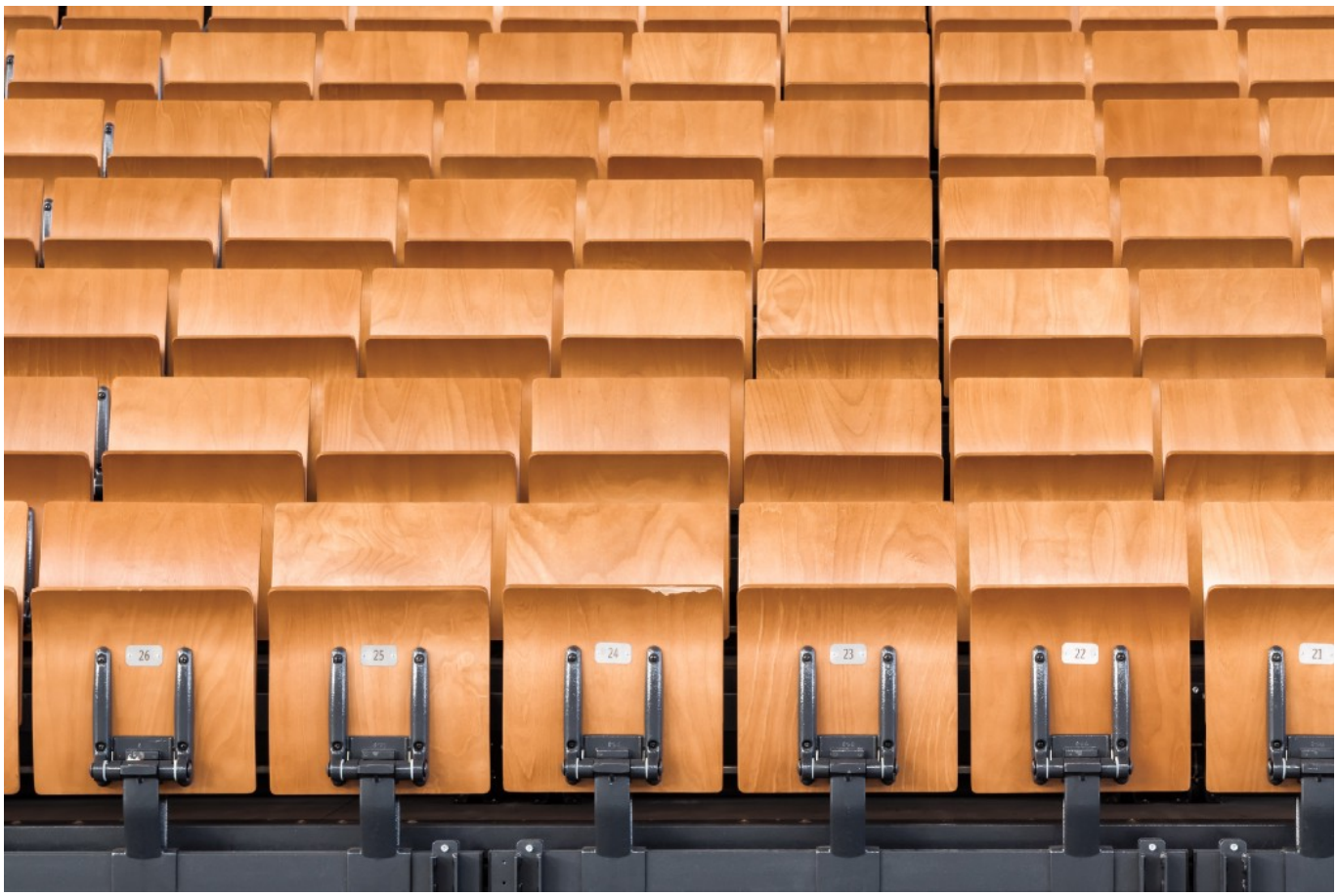

Photo by Paweł Czerwiński on Unsplash

Shallow copy arrays using `Array.from()`

4. Another method to copy a JavaScript array is using `Array.from()`, which will also make a shallow copy, as shown in this example:

```
const emojiArray = ["🤓", "😋", "😇"]
console.log(...emojiArray) // 🤓 😋 😇

// The assignment operator = will only copy the reference to the original array
const emojiArrayNotCopied = emojiArray
console.log(emojiArrayNotCopied === emojiArray) // true -- Both variables reference the original

// Make a shallow copy using Array.from:
const emojiArrayShallowCopy = Array.from(emojiArray)
console.log(emojiArrayShallowCopy === emojiArray) // false -- The variables reference different

emojiArray[0] = "😠" // Change a value in the original array

// The assignment operator does not copy the array, so changes to the original will affect the u
console.log(...emojiArray) // 😠 😋 😇
console.log(...emojiArrayNotCopied) // 😠 😋 😇
console.log(...emojiArrayShallowCopy) // 🤓 😋 😇

// Recap: Like other shallow copy methods, Array.from will make a shallow clone for arrays conta
// But for deeply nested arrays that contain other arrays or objects, you need to deep copy, so
```

Shallow copy an array using Array.from is hosted with ♡ by GitHub view raw

If an object or array contains other objects or arrays, shallow copies will work unexpectedly, because nested objects are not actually cloned.

For deeply-nested objects, a deep copy will be needed. I explain why below.

Photo by brabus biturbo on Unsplash

Watch out for the deeply-nested Gotcha!

On the other hand, when JavaScript objects including arrays are deeply nested, the spread operator only copies the first level with a new reference, but the deeper values are still linked together.

```
const nestedArray = [['😉'],['😋'],['😇']]
const nestedCopyWithSpread = [...nestedArray]

nestedArray[0][0] = '😠' // Whoops, a bug

console.log(...nestedArray) // ["😠"] ["😋"] ["😇"]
console.log(...nestedCopyWithSpread) // ["😠"] ["😋"] ["😇"]

// This is a hack to make a deep copy that is not recommended because it will often fail:
const nestedCopyWithHack = JSON.parse(JSON.stringify(nestedArray)) // Make a deep copy
nestedCopyWithHack[0][0] = '😉' // Only this copied array will be changed

console.log(...nestedArray) // ["😠"] ["😋"] ["😇"]
console.log(...nestedCopyWithSpread) // ["😠"] ["😋"] ["😇"]
console.log(...nestedCopyWithHack) // ["😉"] ["😋"] ["😇"]

// A better solution would be using a library like lodash or Ramda's clone() method
```

Watch out for the deeply-nested Gotcha!.js hosted with ♡ by GitHub view raw

To solve this problem requires creating a deep copy, as opposed to a shallow copy. Deep copies can be made using lodash, rfdc, or the R.clone() method from the Ramda functional programming library. I explore deep copies next.

Photo by Landon Martin on Unsplash

What is a deep copy?

For objects and arrays containing other objects or arrays, copying these objects requires a deep copy. Otherwise, changes made to the nested references will change the data nested in the original object or array.

This is compared to a shallow copy, which works fine for an object or array containing only primitive values, but will fail for any object or array that has nested references to other objects or arrays.

Understanding the difference between `==` and `===` can help visually see the difference between shallow and deep copy, as the strict equality operator (`===`) shows that the nested references are the same:

```
const nestedArray = [["😉"], ["😋"], ["😇"]]
const nestedCopyWithSpread = [...nestedArray]

console.log(nestedArray[0] === nestedCopyWithSpread[0]) // true -- Shallow copy (same reference)

// This is a hack to make a deep copy that is not recommended because it will often fail:
const nestedCopyWithHack = JSON.parse(JSON.stringify(nestedArray))

console.log(nestedArray[0] === nestedCopyWithHack[0]) // false -- Deep copy (different reference

// A better solution would be using a library like lodash or Ramda's clone() method
```

Understanding deep copy by using === to compare references.js hosted with ♡ by GitHub view raw

I will cover 5 methods of making a deep copy (or deep clone): lodash, Ramda, a custom function, `JSON.parse()` / `JSON.stringify()`, and `rfdc`.

Photo by mya thet khine on Unsplash

Deep copy with lodash

1. The library lodash is the most common way JavaScript developers make a deep copy. It is surprisingly easy to use:

```
import _ from "lodash" // Import the entire lodash library
// import { clone, cloneDeep } from "lodash" // Alternatively: Import just the clone methods fro

const nestedArray = [["😉"], ["😊"], ["😇"]]
// This array is nested 1 level, though the behavior is the same with any degree of nesting

const notACopyWithEquals = nestedArray
console.log(nestedArray[0] === notACopyWithEquals[0]) // true -- Not a copy (same reference)

const shallowCopyWithSpread = [...nestedArray]
console.log(nestedArray[0] === shallowCopyWithSpread[0]) // true -- Shallow copy (same reference

const shallowCopyWithLodashClone = _.clone(nestedArray)
console.log(nestedArray[0] === shallowCopyWithLodashClone[0]) // true -- Shallow copy (same refe

const deepCopyWithLodashCloneDeep = _.cloneDeep(nestedArray)
console.log(nestedArray[0] === deepCopyWithLodashCloneDeep[0]) // false -- Deep copy (different

// Try to change the reference for the 1st element, won't work for any copy
nestedArray[0] = "😳"

// Try to change the nested reference for the 3rd element:
nestedArray[2][0] = "😈"

console.log(...nestedArray) // 😳 ["😊"] ["😈"]
console.log(...notACopyWithEquals) // 😳 ["😊"] ["😈"]
console.log(...shallowCopyWithSpread) // ["😉"] ["😊"] ["😈"]
console.log(...shallowCopyWithLodashClone) // ["😉"] ["😊"] ["😈"]
console.log(...deepCopyWithLodashCloneDeep) // ["😉"] ["😊"] ["😇"]
```

Lodash's name comes from the library being referenced as an underscore (_), a "low dash" or lodash for short.

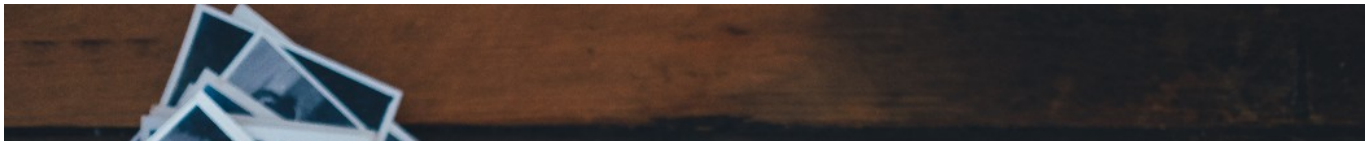

Photo by Annie Spratt on Unsplash

Deep copy with Ramda

2. The functional programming library Ramda includes the `R.clone()` method, which makes a deep copy of an object or array.

```
import R from "ramda" // Import the entire ramda library
// import { clone } from "ramda" // Alternatively: Import just the clone methods from ramda

const nestedArray = [["😉"], ["😊"], ["😇"]]
// This array is nested 1 level, though the behavior is the same with any degree of nesting

const notACopyWithEquals = nestedArray
console.log(nestedArray[0] === notACopyWithEquals[0]) // true -- Not a copy (same reference)

const shallowCopyWithSpread = [...nestedArray]
console.log(nestedArray[0] === shallowCopyWithSpread[0]) // true -- Shallow copy (same reference

const deepCopyWithRamdaClone = R.clone(nestedArray)
console.log(nestedArray[0] === deepCopyWithRamdaClone[0]) // false -- Deep copy (different refer

// Try to change the reference for the 1st element, won't work for any copy
nestedArray[0] = "😳"
// Try to change the nested reference for the 3rd element:
nestedArray[2][0] = "😈"
```

```
20
21  console.log(...nestedArray) // 😳 ["😉"] ["😈"]
22  console.log(...notACopyWithEquals) // 😳 ["😉"] ["😈"]
23  console.log(...shallowCopyWithSpread) // ["😉"] ["😉"] ["😈"]
24  console.log(...deepCopyWithRamdaClone) // ["😉"] ["😉"] ["😇"]
```

Note that `R.clone()` from Ramda is equivalent to `_.cloneDeep()` for lodash, as Ramda does not have a shallow copy helper method.

Photo by Roi Dimor on Unsplash

Deep copy with custom function

3. It is pretty easy to write a recursive JavaScript function that will make a deep copy of nested objects or arrays. Here is an example:

```
1  const deepCopyFunction = (inObject) => {
2    let outObject, value, key
3
4    if (typeof inObject !== "object" || inObject === null) {
```

```
  if (typeof inObject !== "object" || inObject === null) {
    return inObject // Return the value if inObject is not an object
  }

  // Create an array or object to hold the values
  outObject = Array.isArray(inObject) ? [] : {}

  for (key in inObject) {
    value = inObject[key]

    // Recursively (deep) copy for nested objects, including arrays
    outObject[key] = deepCopyFunction(value)
  }

  return outObject
}

let originalArray = [37, 3700, { hello: "world" }]
console.log("Original array:", ...originalArray) // 37 3700 Object { hello: "world" }

let shallowCopiedArray = originalArray.slice()
let deepCopiedArray = deepCopyFunction(originalArray)

originalArray[1] = 0 // Will affect the original only
console.log(`originalArray[1] = 0 // Will affect the original only`)
originalArray[2].hello = "moon" // Will affect the original and the shallow copy
console.log(`originalArray[2].hello = "moon" // Will affect the original array and the shallow c

console.log("Original array:", ...originalArray) // 37 0 Object { hello: "moon" }
console.log("Shallow copy:", ...shallowCopiedArray) // 37 3700 Object { hello: "moon" }
console.log("Deep copy:", ...deepCopiedArray) // 37 3700 Object { hello: "world" }
```

Note that I also need to check for null since the `typeof` null is “object.”

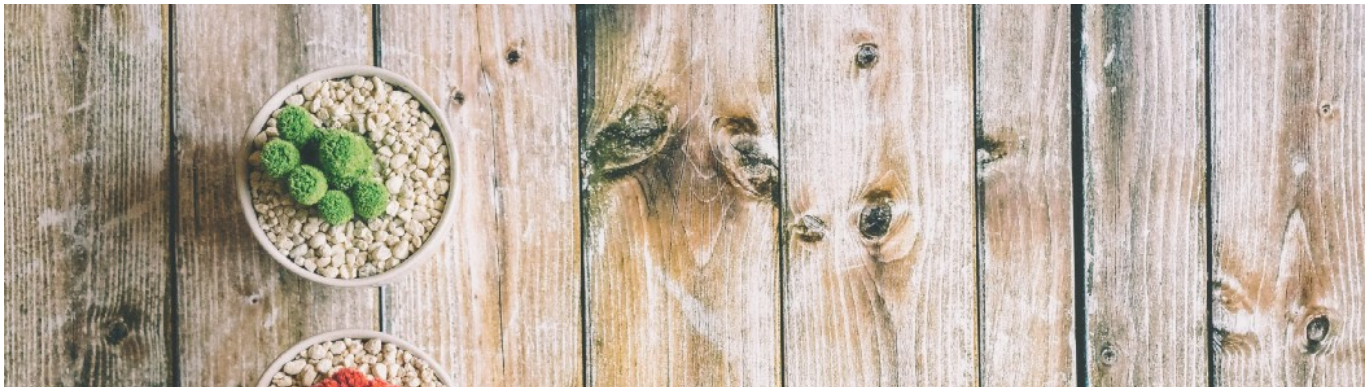

Photo by Scott Webb on Unsplash

Deep copy with JSON.parse/stringify

4. If your data fits the specifications (see below), then `JSON.parse` followed by `JSON.stringify` will deep copy your object.

> "If you do not use `Date`s, functions, `undefined`, `Infinity`, [NaN], RegExps, Maps, Sets, Blobs, FileLists, ImageDatas, sparse Arrays, Typed Arrays or other complex types within your object, a very simple one liner to deep clone an object is: `JSON.parse(JSON.stringify(object))`" — Dan Dascalescu in his StackOverflow answer

To demonstrate some reasons why this method is not generally recommended, here is an example of creating a deep copy using `JSON.parse(JSON.stringify(object))`:

```
// Only some of these will work with JSON.parse() followed by JSON.stringify()
const sampleObject = {
  string: 'string',
  number: 123,
```

```
  boolean: false,
  null: null,
  notANumber: NaN, // NaN values will be lost (the value will be forced to 'null')
  date: new Date('1999-12-31T23:59:59'),  // Date will get stringified
  undefined: undefined,  // Undefined values will be completely lost, including the key containi
  infinity: Infinity,  // Infinity will be lost (the value will be forced to 'null')
  regExp: /.*/, // RegExp will be lost (the value will be forced to an empty object {})
}

console.log(sampleObject) // Object { string: "string", number: 123, boolean: false, null: null,
console.log(typeof sampleObject.date) // object

const faultyClone = JSON.parse(JSON.stringify(sampleObject))

console.log(faultyClone) // Object { string: "string", number: 123, boolean: false, null: null,

// The date object has been stringified, the result of .toISOString()
console.log(typeof faultyClone.date) // string
```

A custom function or the libraries mentioned can make a deep copy without needing to worry about the type of the contents, though circular references will trip all of them up.

Next I discuss a blazing-fast library called `rfdc` that can handle circular references while being as fast as a custom deep copy function.

Photo by Scott Webb on Unsplash

Really fast deep copy? Think `rfdc`

5. For the best performance, the library `rfdc` (Really Fast Deep Clone) will deep copy about 400% faster than lodash's `_.cloneDeep`:

> "`rfdc` clones all JSON types: • `Object` • `Array` • `Number` • `String` • `null`
>
> With additional support for: • `Date` (copied) • `undefined` (copied) • `Function` (referenced) • `AsyncFunction` (referenced) • `GeneratorFunction` (referenced) • `arguments` (copied to a normal object)
>
> All other types have output values that match the output of `JSON.parse(JSON.stringify(o))`." —rfdc Documentation

Using `rfdc` is pretty straight-forward, much like the other libraries:

```
1  const clone = require('rfdc')() // Returns the deep copy function
2  clone({a: 37, b: {c: 3700}}) // {a: 37, b: {c: 3700}}
```

Using rfdc (Really Fast Deep Clone).js hosted with ♡ by GitHub view raw

The `rfdc` library supports all types and also supports circular references with an optional flag that decreases performance by about 25%.

Circular references will break the other deep copy algorithms discussed.

Such a library would be useful if you are dealing with a **large**, complex object such as one loaded from JSON files from 3MB-15MB in size.

Here are the benchmarks, showing `rfdc` is about 400% faster when dealing with such large objects:

> *benchLodashCloneDeep*100: 1461.134ms*
>
> *benchRfdc*100: 323.899ms*
>
> *benchRfdcCircles*100: 384.561ms* — *rfdc Documentation*

- The library `rfdc` (Really Fast Deep Clone) is on GitHub and npm:

davidmarkclements/rfdc

Really Fast Deep Clone. Contribute to davidmarkclements/rfdc development by creating an account on GitHub.

github.com

rfdc

Really Fast Deep Clone

www.npmjs.com

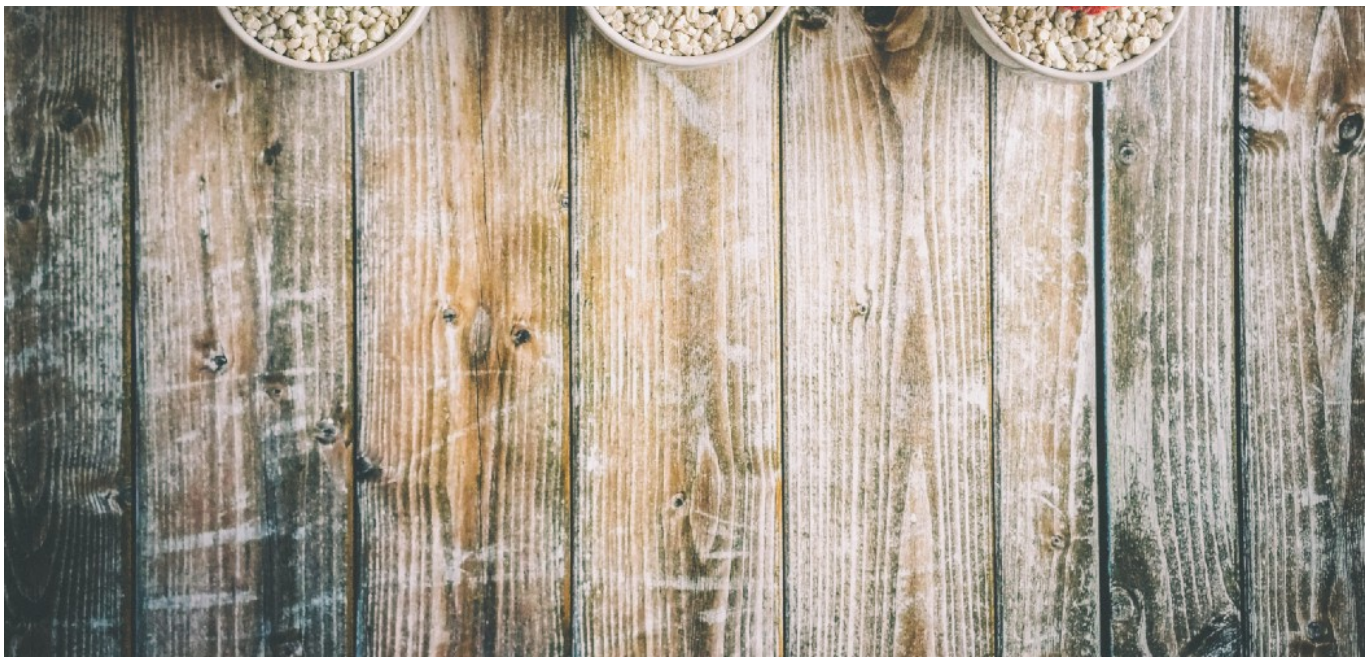

Photo by Scott Webb on Unsplash

Performance of JavaScript Copy Algorithms

Of the various copy algorithms, the shallow copies are the fastest, followed by deep copies using a custom function or `rfdc`:

> ***"Deep copy by performance:** Ranked from best to worst*
>
> *Reassignment " = " (string arrays, number arrays — only)*
>
> *Slice (string arrays, number arrays — only)*
>
> *Concatenation (string arrays, number arrays — only)*
>
> *Custom function: for-loop or recursive copy*
>
> *[*Author's note: `rfdc` would be here, as fast as a custom function*]*
>
> *jQuery's* `$.extend`
>
> `JSON.parse` *(string arrays, number arrays, object arrays — only)*
>
> *Underscore.js's* `_.clone` *(string arrays, number arrays — only)*
>
> *Lo-Dash's* `_.cloneDeep` *" — Tim Montague in his StackOverflow answer*

Using `JSON.parse` / `JSON.stringify` creates issues around data types, so `rfdc` is recommended — unless you want to write a custom function.

Photo by Keila Hötzel on Unsplash

Conclusion

It is actually pretty easy to avoid needing to deep copy in JavaScript — if you can just never nest objects and arrays inside each other.

Because in that case — where there is no nesting and the objects and arrays only contain primitive values — making a shallow copy with the spread operator (`…`), `.slice()` , and `.assign()` all work great.

But, in the real world, where objects have arrays inside them, or vice versa, then a deep copy will need to be used. I recommend `rfdc` for deep clones.

(Note that some may also suggest using `JSON.stringify()` followed by `JSON.parse()` , but that is not a reliable way to make a deep copy.)

Now get out there and deep copy some nested objects!

Photo by Kristina Evstifeeva on Unsplash

Further reading

- Alligator.io has a great article on deep cloning using lodash:

Deep Cloning Objects In JavaScript (And How It Works)

If you plan on coding with JavaScript, you need to understand how objects work. They're one of the most important...

alligator.io

- Samantha Ming explains shallow and deep copies on dev.to:

How to Deep Clone an Array in JavaScript

There are 2 types of array cloning: shallow & deep. Shallow copies only cover

the 1st level of the array and the rest...

dev.to

- Peter Tasker's article on dev.to generated many comments, including an explanation of when `JSON.parse(JSON.stringify(obj))` fails:

Best way to copy an object in JavaScript?

So I'm always looking for a way to use vanilla JS whenever possible these days, and I discovered that deep copying an...

dev.to

- James Dorr covers the need to copy when adding to an array to React State (since React State is immutable) in JavaScript in Plain English:

.setState() Not Working — Shallow Copy vs Deep Copy & Lodash

Shallow Copy vs Deep Copy & Lodash

Shallow Copy vs Deep Copy & Lodashmedium.com

- Ramón Miklus uses immutability-helper, a library used to maintain the immutability of data, instead of lodash to deep copy on Codementor:

Deep copying an object in JavaScript | Codementor

Using the spread syntax or Object.assign() is a standard way of copying an object in JavaScript. Both methdologies can...

www.codementor.io

- Speaking of immutability, Gabriel Lebec has a great guide in dev.to:

This article presents four different techniques to immutably update data structures in JavaScript: Many of the code...

dev.to

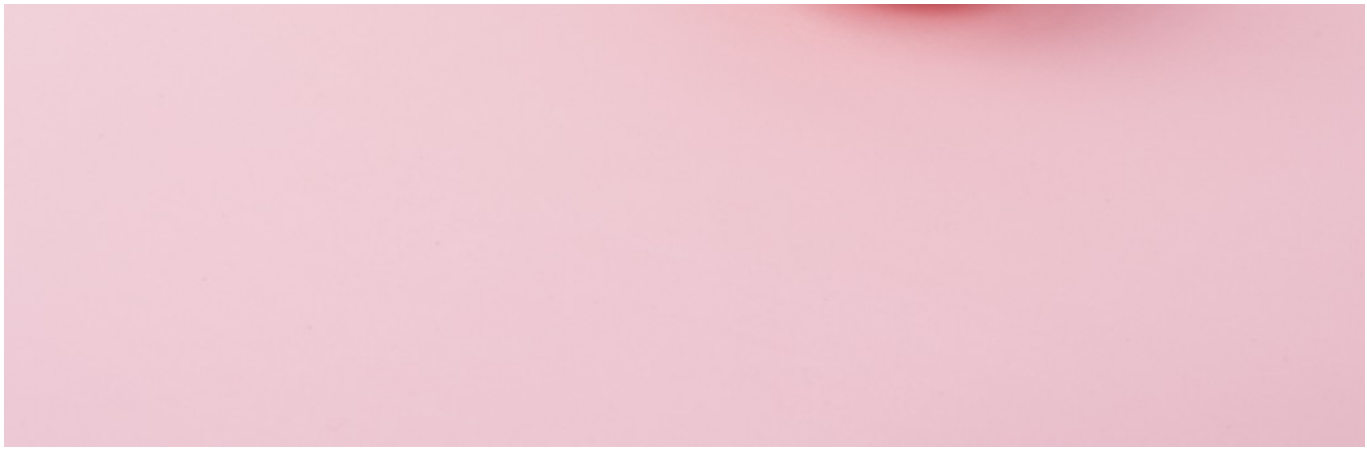

Photo by 青 晨 on Unsplash

JavaScript Programming Technology Software Development Web Development