mdn web docs_

Shallow copy

A **shallow copy** of an object is a copy whose properties share the same references (point to the same underlying values) as those of the source object from which the copy was made. As a result, when you change either the source or the copy, you may also cause the other object to change too. That behavior contrasts with the behavior of a deep copy, in which the source and copy are completely independent.

More formally, two objects `o1` and `o2` are shallow copies if:

1. They are not the same object (`o1 !== o2`).
2. The properties of `o1` and `o2` have the same names in the same order.
3. The values of their properties are equal.
4. Their prototype chains are equal.

See also the definition of *structural equivalence*.

The copy of an object whose properties all have primitive values fits the definition of both a deep copy and a shallow copy. It is somewhat useless to talk about the depth of such a copy, though, because it has no nested properties and we usually talk about deep copying in the context of mutating nested properties.

For shallow copies, only the top-level properties are copied, not the values of nested objects. Therefore:

- Re-assigning top-level properties of the copy does not affect the source object.
- Re-assigning nested object properties of the copy does affect the source object.

In JavaScript, all standard built-in object-copy operations (spread syntax, `Array.prototype.concat()`, `Array.prototype.slice()`, `Array.from()`, and `Object.assign()`) create shallow copies rather than deep copies.

Consider the following example, in which an `ingredientsList` array object is created, and then an `ingredientsListCopy` object is created by copying that `ingredientsList` object.

JS

```js
const ingredientsList = ["noodles", { list: ["eggs", "flour", "water"] }];

const ingredientsListCopy = Array.from(ingredientsList);
console.log(ingredientsListCopy);
// ["noodles",{"list":["eggs","flour","water"]}]
```

Re-assigning the value of a nested property will be visible in both objects.

JS

```js
ingredientsListCopy[1].list = ["rice flour", "water"];
console.log(ingredientsList[1].list);
// Array [ "rice flour", "water" ]
```

Re-assigning the value of a top-level property (the `0` index in this case) will only be visible in the changed object.

JS

```js
ingredientsListCopy[0] = "rice noodles";
console.log(ingredientsList[0]); // noodles
console.log(JSON.stringify(ingredientsListCopy));
// ["rice noodles",{"list":["rice flour","water"]}]
console.log(JSON.stringify(ingredientsList));
// ["noodles",{"list":["rice flour","water"]}]
```

See also

- Related glossary terms:
  - Deep copy