Back to Week 1

JS | Value vs Reference and Mutable Data Types

SELF GUIDED

Learning goals

After this lesson, you will be able to:

- understand how primitives are passed and compared,
- understand how non-primitives (objects and arrays) are passed and compared,
- understand what does mutation mean and what are ways to avoid it,
- know how to use mutating vs. non-mutating methods when adding and removing elements from arrays,
- use spread operator,
- use object destructuring.

Primitives - passed (copied) by `value`

As we know, primitives (strings, numbers, booleans, ...) are immutable data types because, once they are created, we can't change them.

Primitive data types are stored and copied by value, which means two values are equal if they have the same value.

```
let price1 = 20.99;
let price2 = 20.99;
console.log(price1 === price2); // => true

let name1 = 'Ana';
let name2 = 'Ana';
console.log(name1 === name2); // => true
```

Explain this code

In above cases, we can see **two independent variables** and **comparing** them we can see **they are the same because their values are exactly the same**.

```
let price1 = 20.99;
let price2 = price1;
console.log(price1 === price2); // => true
```

Explain this code

The same situation here: `price2` is a new variable and it got the same value as the `price1` (**the value of a variable `price1` is copied into the variable `price2`**) but they are two independent variables, they exist separately in the memory.

Non-primitives - passed (copied) by `reference`

However, the situation is a bit different when working with **arrays** and **objects**.

> Object and array variables don't hold the value of a specific object or array (since what exactly is their value, right?), but instead, they hold the *"address in memory"* which is the **reference** to that object or array.

Now we can ask - if we copy an object or an array, what exactly we are copying? The correct answer is: **we are copying the reference to that object or array**.
Now we can think in this direction: so when we copy a variable that holds string or number or any other primitive value, we are creating a new variable that has the same value.

What is then created when we copy an object or an array if we don't copy its value but instead just the reference to it? The correct answer is: **the new object or array is created BUT it still references the original** (which we copied in the first place). And now, the most important thing for us to remember is: **since both objects or arrays are pointing to the same address in the memory (have the same reference), changes in one will cause the same changes in the other one as well**.

Let's take a look:

```
const book1 = {
  author: 'Charlotte Bronte',
};

const book2 = book1; // => copy the book1 into the new object - book2

console.log(book1); // => { author: 'Charlotte Bronte' }
console.log(book2); // => { author: 'Charlotte Bronte' }

// CHANGE THE VALUE OF AUTHOR PROPERTY IN BOOK1:
book1.author = 'Jane Austen';

// BOTH ARE CHANGED
console.log(book1); // => { author: 'Jane Austen' }
console.log(book2); // => { author: 'Jane Austen' }

// CHANGE THE VALUE OF AUTHOR PROPERTY IN BOOK2:
book2.author = 'Edith Wharton';

// BOTH ARE CHANGED
console.log(book1); // => { author: 'Edith Wharton' }
console.log(book2); // => { author: 'Edith Wharton' }
```


We can see that both variables reference to the same object, which leads us to the conclusion - there's only one object but there are two variables pointing in it..

Okay, we saw how the objects are copied, now let's see how we can compare

them.

> **Two variables are the same only if they reference the same object/array**.

```
// object:
const book1 = {
  author: 'Charlotte Bronte',
};
const book2 = book1; // => copy the book1 into the new object - book2
console.log(book1 === book2); // => true

// array:
const students = ['Ana', 'John', 'Fabio'];
const ironhackers = students;
console.log(students === ironhackers); // => true
```

Explain this code

However, if two objects or arrays look completely the same, but they don't reference the same object/array, they are not the same.

```
// objects:
const book1 = {
  author: 'Charlotte Bronte',
};
const book2 = {
  author: 'Charlotte Bronte',
};

console.log(book1 === book2); // => false

// arrays:
const students = ['Ana', 'John', 'Fabio'];
const ironhackers = ['Ana', 'John', 'Fabio'];

console.log(students === ironhackers); // => false
```

Explain this code

You can use equality `==` or strict equality `===` operators when comparing objects or arrays, they work totally the same in this case.

How to copy an object

If we want to create an independent copy of an object, we should use **`Object.assign()`**, **`for ... in`** loop combined with the initialized empty object or **`JSON.parse(JSON.stringify())`**.

- **Object.assign()**

```
const book1 = {
  author: 'Charlotte Bronte',
};

const book2 = Object.assign({}, book1);

console.log(book2); // => { author: "Charlotte Bronte" }
console.log(book1 === book2); // => false
```

Explain this code

As we can see the objects are the same but they don't reference the same object so changes in one wouldn't cause changes in the other one:

```
book1.author = 'Emily Bronte';
console.log(book1); // => { author: 'Emily Bronte' }
console.log(book2); // => { author: 'Charlotte Bronte' }
```

Explain this code

⚠️ BUT there's one very important thing we have to keep in our mind: *Object.assign()* creates so-called **shallow copy** since all **nested properties still be copied by reference**.

```js
const book1 = {
  author: 'Charlotte Bronte',
  publishers: [
    {companyName: 'AB'},
    {companyName: 'CD'}
  ]
}

const book3 = Object.assign({}, book1);

book1.publishers[0] = {
  companyName: 'Super Cool Company', // => here we changed the name of
  // the 1st publisher in the original (book1)
};

book1.author = 'Test Test'; // => here we changed the author name in the original (book1)

console.log(book3);
// console:
//  { author: 'Charlotte Bronte', // => THIS DIDN'T CHANGE
//    publishers: [
//        { companyName: 'Super Cool Company' }, // => THIS IS CHANGED SINCE IT'S COPIED BY REFERENCE
//        { companyName: 'CD' }
//    ]
//  }
```

Explain this code

So you have to be super careful when using *Object.assign()* since all nested properties will still be connected to the original object and all the changes in the original object's nested properties will affect the copy's nested properties as well.

The solution is using the other option:

- `for ... in` loop combined with initialized empty object - **Deep Cloning**

```
const book1 = {
  author: 'Charlotte Bronte',
};

const book4 = {}; // => INITIALIZED EMPTY OBJECT

for (let prop in book1) {
  book4[prop] = book1[prop];
}

book1.author = 'William Shakespeare'; // => changed the original

console.log(book1); // => { author: 'William Shakespeare' } ==> changed
console.log(book4); // => { author: 'Charlotte Bronte' } ==> DID N'T CHANGE
```


Let’s now take a look to see how this will look like in objects with nested properties. In this case, since we are looping through all the properties of the object, we can check the type of each of them and if the property has type object, then we have to clone it the same way as the parent object.
The best approach would be to create a function that will give us back a copied object with all its properties, including the nested ones. Our example is based on the following **code**:

```
const book1 = {
  author: 'Charlotte Bronte',
  publishers: [
    (publisher1 = {
      companyName: 'AB',
    }),
    (publisher2 = {
      companyName: 'CD',
    }),
  ],
};

function cloneObject(object) {
  let clone = {};
  for (let prop in object) {
    if (object[prop] != null && typeof object[prop] == 'object')
```

```
    {
17          clone[prop] = cloneObject(object[prop]);
18        } else {
19          clone[prop] = object[prop];
20        }
21      }
22      return clone;
23    }
24
25    let book4 = cloneObject(book1); // call the function and create
      the copy => book4
26
27    book1.publishers[0] = {
28      companyName: 'Super Cool Company', // => change the deep prope
      rty of the book1
29    };
30    book1.author = 'William Shakespeare'; // change the property of
      the book1
31    console.log(book1);
32    console.log(book4);
```

Explain this code

If you check your console, you will see that the changes in `book1` object don't affect `book4`.

- **`JSON.parse(JSON.stringify())` - Deep Copy**
  Later in the course we will be talking about JSON object, but now here is short break down:
- The `JSON.stringify()` method converts a JavaScript object or value to a JSON string.
- The `JSON.parse()` method parses a JSON string, constructing the JavaScript value or object described by the string.

Don't spend your time at this point trying to understand how *JSON.parse()* nor *JSON.stringify()* work - you will know it very soon. For now, just be aware of this approach when you need to copy objects in JavaScript.

```
const book1 = {
  author: 'Charlotte Bronte',
  publishers: [
    {companyName: 'AB'},
    {companyName: 'CD'}
  ]
}

const book5 = JSON.parse(JSON.stringify(book1)); // => create the copy: book5

book1.publishers[0] = {
  companyName: 'Super Cool Company', // => change the deep property of the book1
};
book1.author = 'William Shakespeare'; // change the property of the book1
console.log(book1);
console.log(book5);
```

Explain this code

If you check your console, you will see that none of the changes in `book1` didn't affect change in the `book5`. Easy and clean!

How to copy an array

There are a couple of different ways to create a copy of an array.
We will take the array of students as our base:

```
const students = ['Ana', 'John', 'Fabio'];
```

Explain this code

- **ES6 spread operator** - `[...array]` - shallow copy

  One of the latest additions to Javascript was *spread operator* and, amongst the rest, it can be used to make a copy of an array. We will be covering ma bit more in depth the spread operator a bit later, but feel free to do some research now when you know one use cas. 😉

```
const ironhackers = [...students];
students.push('Sandra');

console.log(students); // => [ 'Ana', 'John', 'Fabio', 'Sandra' ]
console.log(ironhackers); // => [ 'Ana', 'John', 'Fabio' ]
```

Explain this code

- **.slice()** - shallow copy

```
const ironhackers = students.slice();
```

Explain this code

Now console.log() both arrays, students and ironhackers, and you will notice that changes in the original array (*students*) don't affect the copied array (*ironhackers*).

- **.concat()** - shallow copy

```
const ironhackers = [].concat(students);
```

Explain this code

Console both arrays and compare them.

- **for loop** - shallow copy

```js
function cloneArray(array) {
  let arrCopy = [];
  for (let i = 0; i < array.length; ++i) {
    arrCopy[i] = array[i];
  }
  return arrCopy;
}

const ironhackers = cloneArray(students); // => invoke function and assign result to variable "ironhackers"
students.push('Sandra');

console.log(students); // => [ 'Ana', 'John', 'Fabio', 'Sandra' ]
console.log(ironhackers); // => [ 'Ana', 'John', 'Fabio' ]
```

✨ Explain this code

Using the shallow copy approach is not necessarily bad, it will work perfectly if the array is not multidimensional.

- `JSON.parse(JSON.stringify())` - Deep Copy

If you really **need a deep copy of an array**, the best approach is using the same **`JSON.parse(JSON.stringify())`** as we did for the objects:

```js
// multidimensional array
const students = [
  ['Ana', 'John', 'Fabio'],
  ['Alex', 'Mike', 'Vero'],
];

// case 1: using spread operator
const ironhackers = [...students];
students[0].push('Sandra');

console.log(students); // [ [ 'Ana', 'John', 'Fabio', 'Sandra' ],
// [ 'Alex', 'Mike', 'Vero' ] ]
console.log(ironhackers); // [ [ 'Ana', 'John', 'Fabio', 'Sandra' ],
```

```
// [ 'Alex', 'Mike', 'Vero' ] ]

// case 2: using JSON.parse(JSON.stringify())
const ironhackers = JSON.parse(JSON.stringify(students));
students[0].push('Sandra');

console.log(students); // [ [ 'Ana', 'John', 'Fabio', 'Sandra' ],
// [ 'Alex', 'Mike', 'Vero' ] ]
console.log(ironhackers); // [ [ 'Ana', 'John', 'Fabio' ], [ 'Alex', 'Mike', 'Vero' ] ]
```

If you take a look at case 1, you can see that changes in the original array cause changes in the copy of the array.
As you can see, using the second approach (`JSON.parse(JSON.stringify())`) the changes in the original array didn't cause changes in the copied array!

Mutable Data Types

At this point, we are pretty sure that everyone understands that arrays and objects are mutable.

> We have to avoid situations that might mutate original objects and arrays. Mutability is bad and it's a side effect that needs to be avoided at any cost.

Can you imagine writing the program and not paying attention to mutation - we would have to take care and keep track of enormous numbers of things at the same time, and at the end, this might cost us creating bad and not usable apps.

Arrays and mutability

Mutable methods	Immutable methods (copy)	What they do
`.push()`	`.concat()`	adding
`.unshift()`	`...` ES6 spread operator	adding
`.splice()`	`.slice()`	removing
`.pop()`	`.slice()` and `...` ES6 spread operator	removing
`.shift()`	`.filter()`	removing

Adding elements to arrays

> 👍 The goal is to keep the original array untouched, to prevent mutating.

Since we already covered most of these methods, we will just shortly touch the rest of them.

Mutable methods

- `.push()` - is used to add a new element to the array and the element is added at the end of the array. **This method doesn't create a new array (copy) but rather mutates the original array**.

```
const students = ['Ana', 'John', 'Fabio'];
students.push('Lilian');
console.log(students); // => [ 'Ana', 'John', 'Fabio', 'Lilian'
]
```

✨ Explain this code

- `.unshift()` - is used to add a new element to the beginning of an array. Once again, **the original array is being mutated since using this method it didn't create a new array**.

One way to avoid mutating the original array is using one of the ways we just learned how to create a copy of the array and then working with the copy and making sure the original array is untouched. Here you have to make sure not to use methods that create shallow copies if you work with multidimensional arrays.

```
const students = ['Ana', 'John', 'Fabio'];
const ironhackers = students.slice();
ironhackers.push('Lilian');
console.log(students); // => [ 'Ana', 'John', 'Fabio' ]
console.log(ironhackers); // => [ 'Ana', 'John', 'Fabio', 'Lilian' ]
```

Explain this code

Immutable methods

- `.concat()` - is a method that **returns a new array**. However, it also returns a shallow copy so keep that in mind when working with more complex arrays.
- `ES6 spread operator` - `[...]` is giving us the chance to add elements to the array without mutating the original array.

```
const students = ['Ana', 'John', 'Fabio'];
const ironhackers = [...students, 'Ariel'];

console.log(students); // => [ 'Ana', 'John', 'Fabio' ]
console.log(ironhackers); // => [ 'Ana', 'John', 'Fabio', 'Ariel' ]
```

Explain this code

We also can pass the new element first and in that case, it will be added to the beginning of the array:

```
const students = ['Ana', 'John', 'Fabio'];
const ironhackers = ['Ariel', ...students];

console.log(students); // => [ 'Ana', 'John', 'Fabio' ]
console.log(ironhackers); // => ['Ariel',  'Ana', 'John', 'Fabi
o' ]
```

Explain this code

Removing elements from arrays

Mutable methods

- `.splice()` - is used to remove a number of elements from the array starting at a certain *position (index)*.

```
const students = ['Ana', 'John', 'Fabio'];
students.splice(1, 1);
console.log(students); // => [ 'Ana', 'Fabio' ]
```

Explain this code

- `.pop()` - is a method used to remove elements from the end of the array.

```
const students = ['Ana', 'John', 'Fabio'];
students.pop();
console.log(students); // => [ 'Ana', 'John' ]
```

Explain this code

- `.shift()` - is a method used to remove the first element from the array.

```
const students = ['Ana', 'John', 'Fabio'];
students.shift();
console.log(students); // => [ 'John', 'Fabio' ]
```

Explain this code

Immutable methods

- `.filter()` - will be covered in one of the next lessons.
- `.slice() and ES6 spread operator` - when used together can save us a lot of trouble since the original array is not mutated.

```
const students = ['Ana', 'John', 'Fabio'];
const ironhackers = [...students.slice(0, 1)];

console.log(students); // => [ 'Ana', 'John', 'Fabio' ]
console.log(ironhackers); // => [ 'Ana' ]
```

Objects and mutability

Mutable methods	Immutable methods	What they do
direct addition	`Object.assign()`	adding
—	ES6 spread operator	adding
`delete` operator	object destructuring	removing

We will cover just methods you didn't have a chance to work with.

- `ES6 spread operator` to add properties to objects without mutating it.
-

```
const book = {
  author: 'Charlotte Bronte',
};
const theSameBook = { ...book, pages: 400 };

console.log(book); // => { author: 'Charlotte Bronte' }
console.log(theSameBook); // => { author: 'Charlotte Bronte', pa
ges: 400 }
```

- `Object destructuring` is a way to remove elements from objects without mutating it.

```
const book = {
  author: 'Charlotte Bronte',
  pages: 400,
  publishers: [
    {
      name: 'publisher1',
    },
    {
      name: 'publisher2',
    },
  ],
};

const { author, ...theRest } = book;

console.log(author); // => Charlotte Bronte
console.log(theRest);
// => { pages: 400, publishers: [ { name: 'publisher1' }, { name: 'publisher2' } ] }
```

Explain this code

The same as the `spread`, we will be covering the `rest` operator in one of the later lessons.

Extra Resources

- Object.assign()
- The JSON.stringify() method
- The JSON.parse() method
- Spread Operator
- Deep copying Javascript Arrays

Mark as completed

PREVIOUS

← LAB | Greatest movies of all time

NEXT

JS | OOP - objects, methods and the 'this' keyword →