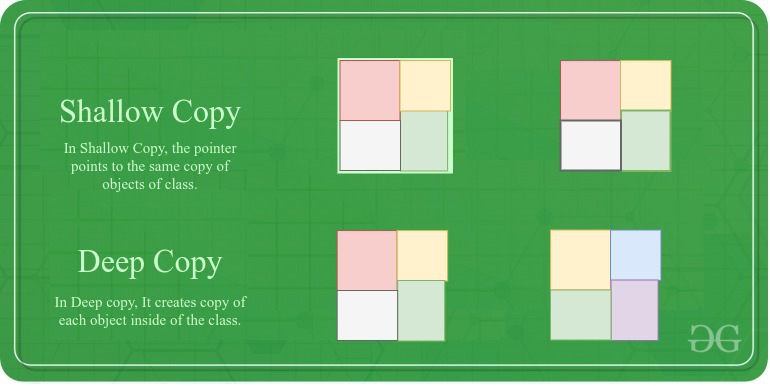
JavaScript Concepts:

1. Shallow copy and Deep copy:

**Shallow Copy:**Shallow repetition is quicker. However, it’s “lazy” it handles pointers and references. Rather than creating a contemporary copy of the particular knowledge the pointer points to, it simply copies over the pointer price. So, each the first and therefore the copy can have pointers that reference constant underlying knowledge.

**Deep Copy:**Deep repetition truly clones the underlying data. It is not shared between the first and therefore the copy.



Below is the tabular Difference between the Shallow Copy and Deep Copy:

| Shallow Copy | Deep Copy |
| --- | --- |
| Shallow Copy stores the references of objects to the original memory address. | Deep copy stores copies of the object’s value. |
| Shallow Copy reflects changes made to the new/copied object in the original object. | Deep copy doesn’t reflect changes made to the new/copied object in the original object. |
| Shallow Copy stores the copy of the original object and points the references to the objects. | Deep copy stores the copy of the original object and recursively copies the objects as well. |
| Shallow copy is faster. | Deep copy is comparatively slower. |

Example:

#### **Primitive data types**

**Primitive data types include the following:**

* Number — e.g. 1
* String — e.g. 'Hello'
* Boolean — e.g. true
* undefined
* null

When you create these values, they are tightly coupled with the variable they are assigned to. They only exist once. That means you do not really have to worry about copying primitive data types in JavaScript. When you make a copy, it will be a real copy. Let’s see an **example**:

const a = 5

let b = a // this is the copy

b = 6

console.log(b) // 6

console.log(a) // 5

By executing b = a , you make the copy. Now, when you reassign a new value to b, the value of b changes, but not of a.

#### **Composite data types — Objects and Arrays**

Now, if we make a copy b = a , and change some nested value in b, it actually changes a’s nested value as well, since a and b actually point to the same thing.

**Example**:

const a = {

en: 'Hello',

de: 'Hallo',

es: 'Hola',

pt: 'Olà'

}

let b = a

b.pt = 'Oi'

console.log(b.pt) // Oi

console.log(a.pt) // Oi

### Objects

There are multiple ways to make copies of objects, especially with the new expanding and improving JavaScript specification.

#### **Spread operator**

Introduced with ES2015, this operator is just great, because it is so short and simple. It ‘spreads’ out all of the values into a new object. You can use it as follows:

const a = {

en: 'Bye',

de: 'Tschüss'

}

let b = {...a}

b.de = 'Ciao'

console.log(b.de) // Ciao

console.log(a.de) // Tschüss

#### Object.assign

This was mostly used before the spread operator was around, and it basically does the same thing. You have to be careful though, as the first argument in the Object.assign() method actually gets modified and returned. So make sure that you pass the object to copy at least as the second argument. Normally, you would just pass an empty object as the first argument to prevent modifying any existing data.

const a = {

en: 'Bye',

de: 'Tschüss'

}

let b = Object.assign({}, a)

b.de = 'Ciao'

console.log(b.de) // Ciao

console.log(a.de) // Tschüss

#### Pitfall: Nested Objects

As mentioned before, there is one big caveat when dealing with copying objects, which applies to both methods listed above. When you have a nested object (or array) and you copy it, nested objects inside that object will not be copied, since they are only pointers / references. Therefore, if you change the nested object, you will change it for both instances, meaning you would end up doing a **shallow copy again**. Example:// BAD EXAMPLE

const a = {

foods: {

dinner: 'Pasta'

}

}

let b = {...a}

b.foods.dinner = 'Soup' // changes for both objects

console.log(b.foods.dinner) // Soup

console.log(a.foods.dinner) // Soup

To make a **deep copy of nested objects**, you would have to consider that. One way to prevent that is manually copying all nested objects:

const a = {

foods: {

dinner: 'Pasta'

}

}

let b = {foods: {...a.foods}}

b.foods.dinner = 'Soup'

console.log(b.foods.dinner) // Soup

console.log(a.foods.dinner) // Pasta

In case you were wondering what to do when the object has more keys than only foods , you can use the full potential of the spread operator. When passing more properties after the ...spread , they overwrite the original values, for example const b = {...a, foods: {...a.foods}} .

**Immediately called function and not :**

(function(){

            console.log("Hello I am Immedialty called funciton");

        })();

        function sayhello() {

            console.log("Hello I am not Immedialty called funciton");

        }

const array = ["one", "two", "three", "four", "five"];

const age = 18 + ""; // converting number to string

age;

typeof(age);

Array Helper Methods:

https://dev.to/epicosity/javascript-array-helper-methods-40mp

1. forEach
2. map
3. filter
4. find
5. every
6. some
7. reduce
8. forEach()
9. // using a for loop
10. const numbers = [1, 2, 3, 4, 5];
11. for (let i = 0; i < numbers.length; i++){
12. console.log(numbers[i])
13. }
14. // Output
15. // 1
16. // 2
17. // 3
18. // 4
19. // 5
20. Let’s refactor this using a classic function declaration.
21. numbers.forEach(function(number){
22. console.log(number);
23. });
24. // 1
25. // 2
26. // 3
27. // 4
28. // 5

// using forEach with arrow

numbers.forEach( number => console.log(number));

Remember — the forEach callback is that it can also take in a second parameter, index. This will keep track of the current index of the number.

numbers.forEach((number, index) => console.log(`Number: ${number} is at index: ${index}`));

// Number: 1 is at index: 0

// Number: 2 is at index: 1

// Number: 3 is at index: 2

// Number: 4 is at index: 3

// Number: 5 is at index: 4