Pass By Value and Pass By Reference in JavaScript

Pass By Value: (Applies to Primitive Data Types)

Primitive data types (`string` , `number` , `boolean` , `null` , `undefined` , `BigInt` , `Symbol`) are **immutable** and stored directly in memory. When assigned to another variable, only the **value** is copied, not the reference.

```
let a = 5;
let b = a;
b = 10;

console.log(a); // Output: 5
console.log(b); // Output: 10
```

📌 Key Concept:

- Changes made to `b` do not affect `a` since they hold independent copies.

Pass By Reference: (Applies to Non-Primitive Data Types)

Objects, arrays, and functions are stored **by reference** in memory. Assigning them to another variable does **not create a copy**; instead, it passes the **reference** to the same memory location.

```
let obj1 = { name: "Abdullah" };
let obj2 = obj1;

obj2.name = "Shaik";

console.log(obj1); // Output: { name: "Shaik" }
console.log(obj2); // Output: { name: "Shaik" }
```

📌 Key Concept:

- Since `obj1` and `obj2` share the same reference, modifying `obj2` also modifies `obj1` .

Cloning Objects to Avoid Reference Issues

If we want to create a copy of an object **without affecting the original**, we must **clone** it.

✅ Shallow Cloning

Shallow cloning works for **single-level objects**, but nested objects are still passed by reference.

Solution 1: Using `Object.assign()`

```
let clonedObj = Object.assign({}, obj1);  // Creates a new object with copied propertie
```

Solution 2: Using Spread Operator (`...`)

```
let clonedObj = { ...obj1 }; // Creates a new object with copied properties
```

⚠️ Limitation:

- These methods **do not deep copy** nested objects.

🔍 The Problem with Shallow Cloning

```
let obj = {
    a: 'a',
    b: 'b',
    c: { deep: 'try and copy' },
};

// Shallow clones
let clone1 = Object.assign({}, obj);
let clone2 = { ...obj };

// Modify the nested object
obj.c.deep = 'hahaha';

console.log(clone1.c.deep); // Output: "hahaha" (unexpected)
console.log(clone2.c.deep); // Output: "hahaha" (unexpected)
```

📌 **Key Issue:**

- The nested object (`c`) is still referenced, meaning changes in `obj` affect both `clone1` and `clone2` .

✅ Deep Cloning (Solving Nested Object Reference Issue)

To **fully copy nested objects**, we need **deep cloning**.

Solution 1: Using `JSON.parse(JSON.stringify(obj))`

```
let deepClone = JSON.parse(JSON.stringify(obj));
```

✅ **Pros:**

- Simple and effective for deeply nested objects.

⚠️ **Cons:**

- **Loses methods & special types** (e.g., `Date` , `RegExp` , `Map` , `Set`).

Solution 2: Using a Recursive Deep Freeze Function

If the object has methods or special types, we can manually deep freeze it.

```
function deepFreeze(obj) {
    Object.keys(obj).forEach(key => {
        if (typeof obj[key] === "object" && obj[key] !== null) {
            deepFreeze(obj[key]);
        }
    });
    return Object.freeze(obj);
}

const deepUser = {
    name: "Charlie",
    details: { age: 40, city: "Los Angeles" }
};

deepFreeze(deepUser);
deepUser.details.age = 45; // No effect
console.log(deepUser.details.age); // Output: 40
```

✅ Pros:

- Ensures immutability even for nested objects.
- Works well for state management (e.g., Redux).

⚠️ Cons:

- Still doesn't preserve methods, dates, or complex objects.

Summary

Concept	Behavior
Pass by Value	Copies the actual value (primitives: `string` , `number` , etc.)
Pass by Reference	Copies the reference, not the value (objects, arrays)
Shallow Copy	Only copies top-level properties (nested objects are still references)
Deep Copy	Fully copies all properties, including nested objects

Concept	Behavior
Best Deep Clone Method	`JSON.parse(JSON.stringify(obj))` (for simple objects) or recursive deep clone for complex objects

✅ Best Practices

- Use **shallow cloning** (`{ ...obj }` or `Object.assign()`) when nested properties don’t need to change.
- Use **deep cloning** (`JSON.parse(JSON.stringify(obj))`) when full object independence is required.
- Use **deep freeze** when immutability is necessary.

By understanding pass by value vs. pass by reference, and how to properly clone objects, you can prevent unintended side effects and write more predictable JavaScript code. 🚀