

Deep vs Shallow Copy in JavaScript

This document explores the critical concepts of deep and shallow copying in JavaScript. Understanding these patterns is essential for writing bug-free code when working with complex data structures such as objects, arrays, and nested collections.

Table of Contents

- 1. Introduction
- 2. Shallow Copy
 - Definition
 - Mechanics
 - Examples
- 3. Deep Copy
 - Definition
 - Mechanics
 - Examples
- 4. Key Differences
- 5. Common Pitfalls
- 6. Best Practices
- 7. Conclusion

1. Introduction

In JavaScript, objects and arrays are reference types. Assigning one variable to another merely copies the reference, not the actual data. Without proper copying, modifying one reference can inadvertently affect another, leading to unexpected side effects. To avoid such issues, two main copying strategies are employed:

- **Shallow Copy:** Creates a new container but retains references to nested objects or arrays.
- **Deep Copy:** Produces a fully independent clone of the data, including all nested structures.

2. Shallow Copy

Definition

A *shallow copy* duplicates the immediate properties or elements of an object or array but does **not** recursively copy nested objects or arrays. Instead, nested references in the copy point to the same memory locations as in the original.

Mechanics

Shallow copying can be achieved using built-in language constructs or utility functions:

- **`Spread operator (...)`** for arrays and objects
- **`Array.prototype.slice()`** for arrays
- **`Object.assign()`** for objects

How it works

1. A new top-level container (object or array) is created.
2. Primitive values (strings, numbers, booleans) are copied by value.
3. References to nested objects/arrays are copied as pointers, not as new instances.

Examples

```
// 1. Using spread operator for arrays
const originalArr = [1, 2, [3, 4]];
const shallowArr = [...originalArr];
// Primitives copied, nested array shared reference

shallowArr[0] = 9;
console.log(originalArr[0]); // 1 (primitive unaffected)

shallowArr[2][0] = 99;
console.log(originalArr[2][0]); // 99 (nested array mutated)

// 2. Using Object.assign for objects
const originalObj = {
  name: 'Alice',
  address: { city: 'Wonderland', zip: 12345 }
};
const shallowObj = Object.assign({}, originalObj);

shallowObj.name = 'Bob';
console.log(originalObj.name); // 'Alice' (primitive unaffected)

shallowObj.address.city = 'Oz';
console.log(originalObj.address.city); // 'Oz' (nested object mutated)
```

3. Deep Copy

Definition

A *deep copy* recursively duplicates every level of an object or array, ensuring no shared references between the original and the clone. After a deep copy, modifications to any part of the clone have no effect on the original.

Mechanics

Several strategies exist for deep copying:

- `JSON.parse(JSON.stringify(obj))`
- `structuredClone(obj)` (modern browsers, Node.js 17+)
- `Utility libraries`: e.g., `_cloneDeep()` from `Lodash`
- `Custom recursive functions` that traverse and clone each property

How it works

1. The function inspects each property or element.
2. Primitive values are copied directly.
3. For objects/arrays, the function recursively creates new instances and clones their contents.
4. Circular references may require special handling.

Examples

```
// 1. Using JSON.parse and JSON.stringify
const original = { a: 1, b: { c: 2 }, d: [3, 4] };
const deepViaJSON = JSON.parse(JSON.stringify(original));
// Limitations: functions, Dates, undefined lost

deepViaJSON.b.c = 999;
console.log(original.b.c); // 2 (independent clone)

// 2. Using structuredClone (modern approach)
const deepViaStructured = structuredClone(original);
// Supports Dates, Maps, Sets, etc.

deepViaStructured.d.push(5);
console.log(original.d); // [3, 4] (original unaffected)

// 3. Using Lodash's cloneDeep (robust, handles functions, regex, circular references)
// import cloneDeep from 'lodash/cloneDeep';
// const deepViaLodash = cloneDeep(original);
// deepViaLodash.b.c = 42;
// console.log(original.b.c); // still 2
```

4. Key Differences

Aspect	Shallow Copy	Deep Copy
--------	--------------	-----------

--	--	--

Nested References	Shared between original and copy	Fully independent clones
-------------------	----------------------------------	--------------------------

Performance	Faster for large structures	Slower due to recursive cloning
Use Cases	When data is flat or nested immutably	When full isolation of data is required
Complexity	Built-in operators/methods	May require libraries or custom logic

5. Common Pitfalls

1. **Accidental Mutation:** Modifying nested properties in a shallow copy affects the original.
2. **Invalid Deep Cloning via JSON:** Loses special types (Date, RegExp, functions, `undefined`).
3. **Circular References:** Can cause stack overflows in naïve recursive clones.
4. **Performance Overhead:** Deep cloning very large structures can be expensive.

6. Best Practices

- **Prefer shallow copy** when you only need to copy top-level data or when using immutable data patterns.
- **Use `structuredClone`** in supported environments for a reliable deep copy of standard data types.
- **Leverage utility libraries** (e.g., Lodash's `cloneDeep`) when you need robust handling of complex or edge-case data types.
- **Avoid JSON methods** for deep copying if preserving functions, Dates, or other non-JSON-safe types is important.
- **Handle circular references** explicitly if your data may contain them (e.g., using `WeakMap` to track visited objects).

7. Conclusion

Copying objects and arrays correctly is vital to prevent unexpected side effects in JavaScript applications. Understanding when to use shallow vs deep copy—and the trade-offs involved—helps maintain data integrity and application performance. By following best practices and choosing the right tool for the job, you can write safer, more predictable code.