

JavaScript learning path

🔥 Great — now let's move into the next advanced layer of spread/rest:

👉 *Hidden internals, traps with Symbols/Set/Map, and advanced merging patterns that even senior devs sometimes miss.*

⚡ Hidden Internals & Edge-Case Behaviors of Spread/Rest

1. Symbol Properties — Lost in Spread

```
js

const sym = Symbol("secret");
const obj = { a: 1, [sym]: 42 };

const copy = { ...obj };

console.log(copy);    // { a: 1 }
console.log(copy[sym]); // undefined ❌
```

👉 Spread only copies **own enumerable string-keyed properties**. Symbols are skipped.

✅ To keep them:

```
js

const copyWithSymbols = {
  ...obj,
  ...Object.fromEntries(Object.getOwnPropertySymbols(obj).map(sym => [sym, obj[sym]]))
};
```

2. Spread with Set / Map

```
js
```

```
const s = new Set([1, 2, 3]);
console.log([...s]); // [1, 2, 3]

const m = new Map([["a", 1], ["b", 2]]);
console.log([...m]); // [ ['a',1], ['b',2] ]
```

- Spread converts `Set` → Array of values.
- Spread converts `Map` → Array of `[key, value]`.

Pro Trick: Convert Map to Object

```
js

const obj = Object.fromEntries(m);
console.log(obj); // { a: 1, b: 2 }
```

3. Array-Like Objects — Trap

```
js

const arrayLike = { 0: "x", 1: "y", length: 2 };

console.log([...arrayLike]); // ❌ TypeError
```

👉 Spread needs **iterables**, not “array-like”.

✅ Fix with:

```
js

console.log(Array.from(arrayLike)); // ['x','y']
```

4. Duplicate Keys in Object Spread

```
js

const a = { x: 1 };
const b = { x: 2, y: 3 };
const c = { ...a, ...b, x: 99 };
```

```
console.log(c); // { x: 99, y: 3 }
```

👉 Rule: Last occurrence wins.

This makes object merging order-sensitive.

5. Getter Traps

js

```
const obj = {  
  get x() {  
    console.log("getter called");  
    return 42;  
  }  
};  
  
const copy = { ...obj }; // triggers getter!  
console.log(copy.x); // 42
```

👉 Spread calls **getters** at copy-time, not lazy.

This can cause **hidden performance bugs**.

6. Prototype Chain — Ignored

js

```
const proto = { p: 1 };  
const obj = Object.create(proto);  
obj.a = 2;  
  
console.log({ ...obj }); // { a: 2 } (❌ no p)
```

👉 Spread ignores prototype chain — only own props copied.

✅ Use `Object.assign(Object.create(Object.getPrototypeOf(obj)), obj)` to preserve prototype.

7. Rest in Object Destructuring — Hidden Pitfall

js

```
const base = { a: 1, b: 2, c: 3 };
const { a, ...rest } = base;

console.log(rest); // { b: 2, c: 3 }
```

👉 But: non-enumerable + symbol props are lost here too.

8. Performance Trade-Off vs `Object.assign`

- `{ ...obj }` and `Object.assign({}, obj)` look same.
- But **spread is syntactic sugar** → internally calls property copy operations.
- `Object.assign` copies **both string + symbol keys**.
- Spread skips symbols → slight semantic difference.

Benchmark:

- Spread slower (~10–30%) on large objects.
 - Use `Object.assign` if you want symbol safety + possibly better perf.
-

Real-World Pro Patterns

1. Immutable Redux Updates

js

```
return { ...state, user: { ...state.user, active: true } };
```

2. Conditional Props in React

jsx

```
<Button {...(isPrimary && { color: "blue" })} />
```

3. Cloning Safe Objects

js

```
const clone = structuredClone(obj); // better than {...obj}
```

👉 Next big area (if you say **continue**) I'll cover:

- ⚡ Performance deep dive: when spread kills performance & when it's worth it.
- 🔥 Weird interview tricks (e.g., spread with `new`, rest + default params, async args).
- 🧩 Production use-cases where spread/rest is "too costly" and alternatives are better.

Do you want me to go **Performance Deep Dive** next, or **Weird Interview Tricks** first?