JavaScript learning path

Perfect 🚀

Let's move to the **next ES6 core feature** that naturally follows destructuring:

🌟 Spread (`...`) & Rest (`...`) Operators

They look the same (`...`), but **context decides meaning**.

---

1. Spread Operator (`...`)

👉 **Expands (spreads) values** out of an array, string, or object.

Example with Arrays

```js
const arr = [1, 2, 3];
const newArr = [0, ...arr, 4];
console.log(newArr); // [0, 1, 2, 3, 4]
```

- Expands elements of `arr` into the new array.

Example with Strings

```js
console.log([..."hello"]); // ['h', 'e', 'l', 'l', 'o']
```

Example with Objects

```js
const obj1 = { a: 1, b: 2 };
const obj2 = { ...obj1, c: 3 };
console.log(obj2); // { a:1, b:2, c:3 }
```

- **Shallow copy** → nested objects are still references.

---

2. Rest Operator (`...`)

👉 **Collects (packs) values** into an array or object.

Example in Functions

```js
function sum(...nums) {
  return nums.reduce((a, b) => a + b, 0);
}
console.log(sum(1, 2, 3, 4)); // 10
```

- Packs all arguments into `nums`.

Example with Objects

```js
const { a, ...rest } = { a: 10, b: 20, c: 30 };
console.log(rest); // { b: 20, c: 30 }
```

Example with Arrays

```js
const [first, ...others] = [10, 20, 30, 40];
console.log(first);  // 10
console.log(others); // [20, 30, 40]
```

---

3. Key Difference (Golden Rule)

- **Spread = expands** data → used in *creation*.
- **Rest = collects** data → used in *destructuring*.

---

4. Real-World Use Cases

1. **Merging arrays**

```js
const nums = [1, 2];
const all = [...nums, 3, 4];
```

2. **Copying objects**

```js
const copy = { ...original };
```

3. **Immutable updates**

```js
const user = { name: "Alice", age: 20 };
const updated = { ...user, age: 21 }; // doesn't mutate original
```

4. **Variable arguments in functions**

```js
function logAll(...args) {
  console.log(args);
}
logAll("a", "b", "c"); // ["a", "b", "c"]
```

---

5. Interview Gotcha

```js
const obj = { a: 1, b: 2, a: 3 };
console.log(obj); // ?
```

✅ `{ a: 3, b: 2 }`

- Later spread/assignment overwrites earlier ones.

```js
const arr = [..."12345"];
console.log(arr); // ["1","2","3","4","5"]
```

👉 Spread works on **iterables**, not just arrays.

✨ That's the **intro + essentials**.
Now, I can:

1. Dive into **hidden internals & performance trade-offs** of spread/rest, or

2. Show **advanced tricks & corner cases** (like deep copies, merging traps, non-iterables).

👉 Which direction do you want to go first?

🔥 Great choice — let's go **deep into the hidden internals + performance trade-offs of `spread` / `rest`**.
This is where most devs (even seniors) slip up in **interviews & production code**.

🛠 Hidden Internals of Spread & Rest

---

1. How Spread Works Under the Hood

When you do:

```
js

const arr = [1, 2, 3];
const copy = [...arr];
```

What happens internally:

- JS calls the **iterator protocol** of `arr` (via `arr[Symbol.iterator]()`).
- Iterates over values and **copies references** (not deep values).
- Creates a new array instance and pushes those values.

So:

- **Primitive values** → copied safely.
- **Objects/arrays inside** → only the reference is copied.

```
js

const nested = [{ x: 1 }, { y: 2 }];
const copy = [...nested];
copy[0].x = 99;
console.log(nested[0].x); // 99 ❌ (shallow copy!)
```

👉 Spread ≠ deep clone.

---

2. How Rest Works Under the Hood

When you do:

```
js

function foo(a, ...rest) {
  console.log(a, rest);
}
foo(1, 2, 3, 4);
```

- `a` gets `1`.
- `rest` collects leftover args into a **real array** (not the old `arguments` object).
- Unlike `arguments`, `rest`:
  - Supports **array methods** (`map`, `reduce`).
  - Doesn't include extra properties like `callee`.
  - Works better with arrow functions.

---

3. Performance Costs

Spread in Arrays

```
js

const arr = new Array(1e6).fill(0);
console.time("spread");
const copy = [...arr];
console.timeEnd("spread");
```

- **Spread = O(n)** because it must iterate over all elements.
- Faster than `Array.prototype.slice()` in modern engines, but slower than **structured cloning** (`structuredClone`) for deep copies.
- **Big arrays** → avoid repeated spreading inside loops.

Spread in Objects

```
js

const obj = { a: 1, b: 2 };
```

```
const copy = { ...obj };
```

- Internally → enumerates all **own enumerable properties** (like `Object.assign`).
- **Symbols are skipped** unless explicitly included.
- Costlier than `Object.assign` in some engines because spread does extra iteration steps.

Rest in Functions

- Creates a **new array every call** → avoid in performance-critical tight loops.
- Use `arguments` object when ultra-optimizing (rarely needed).
- But for clean, modern code → `rest` is preferred.

---

4. Memory Footprint

- Spread always creates a **new object/array**.
- If used repeatedly in reducers / immutability patterns:

```
js

state = { ...state, key: value }
```

→ Can create **lots of garbage objects** → triggers GC (garbage collector).
In **React**, this is fine because immutability is critical, but in raw JS loops → prefer in-place mutation if safe.

⚡ Advanced Corner Cases & Gotchas

---

1. Non-Iterable Spread

```
js

const notIterable = { a: 1, b: 2 };
// const fail = [...notIterable]; ❌ TypeError
```

- Spread only works on **iterables** (arrays, strings, sets, maps).
- But object spread (`{...obj}`) is a **different mechanism** → property copying.

2. Property Overwrites in Object Spread

```js
const a = { x: 1, y: 2 };
const b = { y: 99, z: 100 };
const merged = { ...a, ...b };
console.log(merged); // { x:1, y:99, z:100 }
```

- Rightmost spread wins (last in order overwrites previous ones).

---

3. Undefined & Null in Spread

```js
console.log({ ...null });  // {}
console.log({ ...undefined }); // {}
```

✅ Objects handle them gracefully.

But:

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

Arrays require an iterable, `null` isn't one.

---

4. Spread in Function Calls

```js
function add(a, b, c) { return a + b + c; }
const nums = [1, 2, 3];
console.log(add(...nums)); // 6
```

- Spread expands array into function args.
- Can mix with normal args:

```js
```

```
add(0, ...nums, 10); // 16
```

---

5. Destructuring with Rest in Objects

```
js

const user = { id: 1, name: "Alice", age: 20 };
const { id, ...details } = user;
console.log(details); // { name: "Alice", age: 20 }
```

- Rest must come **last** in destructuring.
- `{ ...details, id } = user;` ❌ SyntaxError.

🧠 Interview Brain-Benders

Q1:

What's the output?

```
js

const obj1 = { a: 1, b: { c: 2 } };
const obj2 = { ...obj1 };
obj2.b.c = 99;
console.log(obj1.b.c);
```

✅ `99` → because spread is **shallow copy**.

Q2:

What's the output?

```
js

const arr1 = [1, 2, 3];
const arr2 = [...arr1, arr1];
console.log(arr2);
```

✅ `[1, 2, 3, [1, 2, 3]]`

Spread doesn't flatten nested arrays, it just copies the reference.

Q3:

Why is this **bad for performance**?

```js
let arr = [];
for (let i = 0; i < 10000; i++) {
  arr = [...arr, i];
}
```

Because each spread creates a **brand new array copy** ($O(n^2)$ total).

Better:

```js
let arr = [];
for (let i = 0; i < 10000; i++) {
  arr.push(i);
}
```

✅ That's the **deep dive into hidden internals + trade-offs.**

Would you like me to now create a **special section of advanced tricks & pro-use cases** of spread/rest (like functional programming patterns, React optimizations, merging strategies) — or jump to the **next ES6 topic**?