Perfect. Continuing **systematically**, **NotebookLM-ready**, with **NOTES and CODE strictly separated**.

📘 TOPIC: Objects in JavaScript

(Creation • Access • Reference Behavior • Internals • Interview Traps)

🟦 NOTE 1: What an object really is (internal view)

- An object is a **non-primitive reference type**
- Stored in **heap memory**
- Variables hold a **reference (address)** to the object
- Objects are **mutable**
- Objects are key–value collections

👉 Objects are the backbone of JS (arrays, functions, dates = objects)

🟦 NOTE 2: Object creation methods

JavaScript provides multiple ways to create objects:

1. Object literal `{}` (most common)
2. `new Object()`
3. Constructor functions
4. `Object.create()`
5. Classes (syntactic sugar)

🟩 CODE 1: Object literal (recommended)

```js

const user = {
  name: "Anoop",
  age: 22,
  isAdmin: false
};
```

🟦 NOTE 3: Object literal internals

- Keys are stored as **strings (or symbols)**
- Values can be **any type**
- Order of keys is **mostly predictable** (but don't rely on it)

🟩 CODE 2: Key normalization

```js

const obj = {
  1: "one",
```

```
  true: "yes"
};

obj["1"];    // "one"
obj[true];   // "yes"
```

🟦 NOTE 4: Accessing object properties

Two ways:

1. Dot notation
2. Bracket notation

🟩 CODE 3: Access methods

```
js

user.name;      // dot
user["age"];    // bracket
```

🟦 NOTE 5: When bracket notation is REQUIRED

- Dynamic keys
- Keys with spaces or special characters
- Numeric keys

🟩 CODE 4: Bracket-only cases

```
js

const obj = {
  "full name": "Anoop Yadav"
};

obj["full name"]; // works
// obj.full name ❌ syntax error

let key = "age";
user[key]; // dynamic access
```

🟦 NOTE 6: Adding, updating, deleting properties

Objects are mutable → properties can change anytime.

🟩 CODE 5: Mutation operations

```
js
```

```
user.city = "Delhi";   // add
user.age = 23;         // update
delete user.isAdmin;   // delete
```

🟦 NOTE 7: Object reference behavior (CRITICAL)

- Objects are assigned by **reference**
- Copying variable does NOT copy object
- Multiple variables can point to same object

🟩 CODE 6: Reference behavior

```
js

const a = { x: 10 };
const b = a;

b.x = 20;
console.log(a.x); // 20
```

🟦 NOTE 8: Comparison of objects

- Objects are compared by **reference**
- Identical structure ≠ equal

🟩 CODE 7: Object comparison trap

```
js

{} === {};   // false

const o = {};
const p = o;

o === p;     // true
```

🟦 NOTE 9: Shallow copy vs reference copy

- Reference copy → same object
- Shallow copy → new object, same nested references

🟩 CODE 8: Shallow copy examples

```
js

const obj1 = { a: 1, b: { c: 2 } };

const obj2 = { ...obj1 }; // shallow copy
```

```
obj2.b.c = 99;

obj1.b.c; // 99 (shared nested reference)
```

🟦 NOTE 10: Deep copy concept

- Deep copy duplicates **all nested objects**
- JS has no built-in deep copy
- Common methods:
  - `structuredClone()`
  - `JSON.parse(JSON.stringify())` (limitations)

🟩 CODE 9: Deep copy (safe modern way)

```
js

const original = { a: 1, b: { c: 2 } };
const copy = structuredClone(original);

copy.b.c = 100;
original.b.c; // 2
```

🟦 NOTE 11: Object methods

- Functions stored as object properties
- `this` refers to **calling object**

🟩 CODE 10: Object method & `this`

```
js

const user = {
  name: "Anoop",
  greet() {
    console.log(this.name);
  }
};

user.greet(); // Anoop
```

🟦 NOTE 12: `this` trap inside objects

- Arrow functions do NOT have own `this`
- They inherit `this` from lexical scope

🟩 CODE 11: Arrow function trap

```
js
```

```
const obj = {
  name: "Anoop",
  greet: () => {
    console.log(this.name);
  }
};

obj.greet(); // undefined
```

🟦 NOTE 13: Checking property existence

Two ways:

1. `in` operator
2. `hasOwnProperty()`

🟩 CODE 12: Property existence

```
js

"name" in user;            // true
user.hasOwnProperty("age");  // true
```

🟦 NOTE 14: Iterating over objects

- `for...in` → keys
- `Object.keys()` → array of keys
- `Object.values()` → array of values
- `Object.entries()` → key–value pairs

🟩 CODE 13: Object iteration

```
js

for (let key in user) {
  console.log(key, user[key]);
}

Object.entries(user).forEach(([k, v]) => {
  console.log(k, v);
});
```

🟦 NOTE 15: Common interview traps

- Objects are passed by reference
- Shallow copy ≠ deep copy

- Arrow functions don’t bind `this`
- Object comparison checks reference
- Deleting properties affects all references

🟦 FINAL MENTAL MODEL

- Objects live in heap
- Variables store references
- Mutations affect all references
- Methods depend on call-site `this`
- Copying objects requires care

🛑 STOP HERE

Next **recommended continuation**:

1️⃣ Prototype & Prototypal Inheritance (VERY important)
2️⃣ Array methods (`map`, `filter`, `reduce`)
3️⃣ Classes & OOP in JavaScript

Reply with **next** or the **number** you want to continue.