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

📘 TOPIC: Classes & OOP in JavaScript

(Syntax Sugar ● Internals ● Inheritance ● `this` ● Interview Traps)

🟦 NOTE 1: What `class` really is (truth)

- JavaScript **does NOT become class-based**
- `class` is **syntactic sugar** over:
  - constructor functions
  - prototypes
- Internally, JS still uses **prototypal inheritance**

👉 If you understand prototypes, you already understand classes.

🟦 NOTE 2: Why `class` was introduced

- Improve readability
- Familiar syntax for Java / C++ developers
- Reduce boilerplate for inheritance
- But **no new inheritance model**

🟩 CODE 1: Basic class syntax

```js
class Person {
  constructor(name) {
    this.name = name;
  }

  greet() {
    console.log("Hi", this.name);
  }
}

const p = new Person("Anoop");
p.greet();
```

🟦 NOTE 3: What happens internally (IMPORTANT)

When you write a class:

- `constructor` → becomes the constructor function
- Methods → stored on `ClassName.prototype`
- `new`:

1. Creates empty object
2. Links `[[Prototype]]`
3. Binds `this`
4. Executes constructor

🟩 CODE 2: Internal equivalence (mental model)

```js
function Person(name) {
  this.name = name;
}

Person.prototype.greet = function () {
  console.log("Hi", this.name);
};
```

🟦 NOTE 4: Classes are NOT hoisted like functions

- Class declarations are hoisted
- BUT remain in **Temporal Dead Zone**
- Cannot be used before declaration

🟩 CODE 3: Hoisting behavior

```js
// const p = new Person("A"); ❌ ReferenceError

class Person {
  constructor(name) {
    this.name = name;
  }
}
```

🟦 NOTE 5: Strict mode by default

- Code inside classes runs in **strict mode**
- Even without `"use strict"`
- Prevents accidental globals

🟩 CODE 4: Strict mode effect

```js
class Test {
  method() {
    x = 10; // ❌ ReferenceError (no implicit global)
  }
}
```

🟦 NOTE 6: Instance properties vs prototype methods

- Properties defined in constructor → **per instance**
- Methods defined in class body → **shared via prototype**

🟩 CODE 5: Instance vs prototype

```js
class User {
  constructor(name) {
    this.name = name; // instance property
  }

  greet() {
    console.log(this.name); // prototype method
  }
}
```

🟦 NOTE 7: `this` inside classes

- `this` refers to **instance**
- Determined by **call site**
- Same rules as regular functions

🟩 CODE 6: `this` trap

```js
class Demo {
  show() {
    console.log(this);
  }
}

const d = new Demo();
const fn = d.show;

fn(); // undefined (or global in non-strict)
```

🟦 NOTE 8: Fixing `this` issues

Solutions:

- Bind in constructor
- Use arrow functions
- Call via instance

🟩 CODE 7: Binding `this`

```
js

class Demo {
 constructor() {
   this.show = this.show.bind(this);
 }

 show() {
   console.log(this);
 }
}
```

🟦 NOTE 9: Inheritance with `extends`

- `extends` sets prototype chain
- Child prototype → parent prototype
- Enables method reuse

🟩 CODE 8: Basic inheritance

```
js

class Animal {
 speak() {
   console.log("Animal sound");
 }
}

class Dog extends Animal {}

const d = new Dog();
d.speak();
```

🟦 NOTE 10: `super` keyword (deep)

- `super()` → calls parent constructor
- Must be called **before using** `this`
- `super.method()` → calls parent method

🟩 CODE 9: `super` usage

```js
class Animal {
  constructor(type) {
    this.type = type;
  }
}

class Dog extends Animal {
  constructor(name) {
    super("Dog");
    this.name = name;
  }
}
```

🟦 NOTE 11: Method overriding

- Child method with same name
- Overrides parent version
- Parent method still accessible via `super`

🟩 CODE 10: Method override

```js
class Animal {
  speak() {
    console.log("Animal speaks");
  }
}

class Dog extends Animal {
  speak() {
    super.speak();
    console.log("Dog barks");
  }
}

new Dog().speak();
```

🟦 NOTE 12: Static methods

- Belong to class, not instances
- Used for utility logic
- Not accessible via `this` of instance

🟩 CODE 11: Static method

```js
class MathUtil {
  static add(a, b) {
    return a + b;
  }
}

MathUtil.add(2, 3); // 5
```

🟦 NOTE 13: Public fields (modern JS)

- Declared directly in class
- Part of instance
- No prototype involvement

🟩 CODE 12: Public field

```js
class User {
  role = "user";
}
```

🟦 NOTE 14: Private fields (`#`)

- Truly private
- Enforced by language
- Not accessible outside class

🟩 CODE 13: Private field

```js
class Account {
  #balance = 0;

  deposit(amount) {
    this.#balance += amount;
  }
}
```

🟦 NOTE 15: Common interview traps

- Classes are not hoisted like functions

- Methods live on prototype
- `this` rules don't change
- `extends` sets prototype chain
- `class` ≠ classical inheritance

🟦 FINAL MENTAL MODEL

- `class` = cleaner syntax over prototypes
- No new inheritance model
- `new` + prototype = core mechanism
- `extends` wires prototype chain
- `this` still depends on call site
- Use classes for readability, not magic

🛑 STOP HERE

Next **recommended continuation**:

1️⃣ **Array Methods** (`map`, `filter`, `reduce`)
2️⃣ `this` **keyword (ALL rules, very deep)**
3️⃣ **Async JavaScript (callbacks → promises → async/await)**

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