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

📘 TOPIC: `this` Keyword in JavaScript

(Binding Rules ● Call-Site Logic ● Arrow Functions ● Classes ● Interview Traps)

🟦 NOTE 1: What `this` actually is (core truth)

- `this` is **NOT** determined where a function is written
- `this` is determined **when and how a function is called**
- `this` is resolved at **runtime**
- Except for **arrow functions**

👉 **Golden Rule**:

> *"`this` depends on the call site"*

🟦 NOTE 2: `this` is NOT lexical (important)

- `this` does NOT follow lexical scope
- Scope ≠ `this`
- Two functions in same scope can have different `this`

🟩 CODE 1: Same scope, different `this`

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

show();          // global (or undefined in strict)
obj.show();      // obj
```

🟦 NOTE 3: The 5 binding rules of `this` (ORDER MATTERS)

JavaScript resolves `this` using these rules **in priority order**:

1. `new` binding
2. Explicit binding (`call`, `apply`, `bind`)
3. Implicit binding (object call)
4. Default binding
5. Arrow function (lexical)

🟦 NOTE 4: Default binding

- Happens when function is called normally
- Non-strict mode → global object

- Strict mode → `undefined`

🟩 CODE 2: Default binding

```js
function test() {
  console.log(this);
}

test(); // window (non-strict) | undefined (strict)
```

🟦 NOTE 5: Implicit binding (object method call)

- When function is called via object
- `this` → object before the dot

🟩 CODE 3: Implicit binding

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

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

🟦 NOTE 6: Implicit binding LOST (interview trap)

- When method is assigned to variable
- Object context is lost

🟩 CODE 4: Lost binding

```js
const fn = user.show;
fn(); // undefined (or global)
```

🟦 NOTE 7: Explicit binding (`call`, `apply`, `bind`)

- You manually decide `this`
- Highest priority after `new`

🟩 CODE 5: `call`

```js
```

```
function greet(city) {
  console.log(this.name, city);
}

greet.call({ name: "Anoop" }, "Delhi");
```

🟩 CODE 6: `apply`

```js
greet.apply({ name: "Anoop" }, ["Delhi"]);
```

🟦 NOTE 8: `bind` (permanent binding)

- Returns a **new function**
- `this` is permanently fixed
- Cannot be overridden

🟩 CODE 7: `bind`

```js
const bound = greet.bind({ name: "Anoop" }, "Delhi");
bound();
```

🟦 NOTE 9: `new` binding (highest priority)

- `new` creates a new object
- `this` → newly created object
- Overrides all other bindings

🟩 CODE 8: `new` binding

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

const u = new User("Anoop");
console.log(u.name);
```

🟦 NOTE 10: Binding priority proof (INTERVIEW)

🟩 CODE 9: Priority example

```js
```

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

const obj1 = { name: "Obj1" };
const obj2 = { name: "Obj2" };

const bound = show.bind(obj1);
new bound(); // `this` is new object, NOT obj1
```

🟦 NOTE 11: Arrow functions (special rule)

- Arrow functions **do NOT have their own** `this`
- They **capture** `this` **lexically**
- `call`, `apply`, `bind` do NOTHING

🟩 CODE 10: Arrow lexical `this`

```
js


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

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

🟦 NOTE 12: Arrow functions inside m