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

📘 TOPIC: Prototype & Prototypal Inheritance

(How JS Inherits • `[[Prototype]]` • `__proto__` • `prototype` • Interview Traps)

🟦 NOTE 1: JavaScript uses prototypal inheritance (not classical)

- JavaScript does **not** use class-based inheritance internally
- It uses **objects inheriting from other objects**
- This linkage is called the **prototype chain**
- Every object has an internal slot: `[[Prototype]]`

👉 Inheritance = **property lookup delegation**

🟦 NOTE 2: What `[[Prototype]]` is

- `[[Prototype]]` is an **internal reference**
- Points to another object (or `null`)
- Not directly accessible
- Controls property lookup

🟩 CODE 1: Prototype lookup behavior

```js
const parent = { x: 10 };
const child = {};

Object.setPrototypeOf(child, parent);

child.x; // 10 (found via prototype)
```

🟦 NOTE 3: Property lookup algorithm (IMPORTANT)

When accessing `obj.prop`:

1. Check `obj` itself
2. If not found → check `obj.[[Prototype]]`
3. Continue up the chain
4. Stop at `null`
5. If not found → `undefined`

🟩 CODE 2: Prototype chain lookup

```js
```

```
const grandParent = { a: 1 };
const parent = Object.create(grandParent);
const child = Object.create(parent);

child.a; // 1
```

🟦 NOTE 4: `__proto__` (legacy accessor)

- `__proto__` exposes `[[Prototype]]`
- Exists for debugging & compatibility
- Avoid using in production code

🟩 CODE 3: `__proto__` example

```
js

const obj = {};
obj.__proto__ === Object.prototype; // true
```

🟦 NOTE 5: `Object.prototype` (root object)

- Almost all objects inherit from `Object.prototype`
- Provides methods like:
  - `toString`
  - `hasOwnProperty`
  - `valueOf`

🟩 CODE 4: Root prototype

```
js

const obj = {};

obj.toString(); // inherited
```

🟦 NOTE 6: Prototype chain end

- Prototype chain ends at `null`
- `Object.prototype.__proto__ === null`

🟩 CODE 5: Chain termination

```
js

Object.prototype.__proto__ === null; // true
```

🟦 NOTE 7: Constructor functions & `prototype`

- Functions have a `prototype` property
- Used **only when function is called with** `new`
- Determines prototype of created objects

🟩 CODE 6: Constructor + prototype

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

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

const p1 = new Person("Anoop");
p1.greet(); // Hi Anoop
```

🟦 NOTE 8: `prototype` vs `__proto__` (INTERVIEW)

Term	Belongs to	Purpose
`prototype`	Constructor function	Used for inheritance
`__proto__`	Object instance	Points to prototype
`[[Prototype]]`	Internal	Actual linkage

🟩 CODE 7: Relationship proof

```js
p1.__proto__ === Person.prototype; // true
```

🟦 NOTE 9: Inheritance using constructor functions

- Child constructor calls parent constructor
- Child prototype linked to parent prototype

🟩 CODE 8: Constructor inheritance

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

```
Animal.prototype.speak = function () {
  console.log("Animal sound");
};

function Dog(name) {
  Animal.call(this, "Dog");
  this.name = name;
}

Dog.prototype = Object.create(Animal.prototype);
Dog.prototype.constructor = Dog;

const d = new Dog("Tommy");
d.speak(); // Animal sound
```

🟦 NOTE 10: Why `constructor` reset is needed

- `Object.create` replaces prototype
- Constructor reference is lost
- Must be reassigned manually

🟦 NOTE 11: `Object.create()` (pure prototypal inheritance)

- Creates object with specified prototype
- No constructor function involved
- Clean and explicit

🟩 CODE 9: Object.create pattern

```
js

const base = {
  greet() {
    console.log("Hello");
  }
};

const obj = Object.create(base);
obj.greet(); // Hello
```

🟦 NOTE 12: Method shadowing

- If property exists on object itself
- Prototype property is ignored

🟩 CODE 10: Shadowing example

```js
const proto = { x: 10 };
const obj = Object.create(proto);

obj.x = 20;
obj.x; // 20 (shadows prototype)
```

🟦 NOTE 13: Performance considerations

- Deep prototype chains slow lookup
- Prefer composition over deep inheritance
- Cache frequently accessed properties

🟦 NOTE 14: Common interview traps

- All objects have `__proto__`
- Only functions have `prototype`
- Prototype lookup is dynamic
- `class` is syntactic sugar
- Inheritance is delegation, not copying

🟦 FINAL MENTAL MODEL

- Objects delegate to prototypes
- `[[Prototype]]` controls inheritance
- `prototype` is for constructor-created objects
- `__proto__` exposes linkage
- JS inheritance is dynamic and flexible

🛑 STOP HERE

Next **recommended continuation**:

1️⃣ **Classes & OOP in JavaScript (syntax sugar + internals)**
2️⃣ **Array methods (`map`, `filter`, `reduce`)**
3️⃣ **`this` keyword (deep rules across contexts)**

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