Perfect — restarting **cleanly** 👍

This version is **NotebookLM-optimized**, **clean**, and **strictly separates**:

- 🟦 **NOTES (theory / points)**
- 🟩 **CODE (only code, no explanations mixed)**

We will do **ONLY** `console`, nothing else.

📘 TOPIC: `console` (Internal Working)

🟦 NOTE 1: What `console` really is

- `console` is **NOT part of JavaScript (ECMAScript)**
- It is provided by the **runtime environment**
  - Browser → Web API
  - Node.js → Runtime API
- JavaScript language itself does not guarantee `console`

🟩 CODE 1: Accessing console

```js

console.log("Hello");
```

🟦 NOTE 2: Where `console` comes from

- `console` is attached to the **global object**
- It exists **before your JS code starts executing**
- That's why it's always available at the top level

🟩 CODE 2: Global object binding

```js

// Browser
console === window.console; // true

// Node.js
console === global.console; // true
```

🟦 NOTE 3: `console` is just an object

- `console` is an object
- `log`, `warn`, `error`, etc. are **methods**
- These methods are implemented by the host, not JS engine

🟩 CODE 3: Inspecting console

```js
```

```
console.log(console);
```

🟦 NOTE 4: What happens internally on `console.log()`

- JS engine evaluates arguments
- Resolves `console` via scope chain
- Calls host-provided function
- Control moves **outside JS engine**
- Host handles output
- JS execution continues immediately

🟩 CODE 4: Execution continues

```
js

console.log("Before");
console.log("After");
```

🟦 NOTE 5: `console.log` is NOT blocking

- JS execution does **not pause**
- Call stack continues normally
- Only rendering of output is delayed

🟩 CODE 5: Non-blocking behavior

```
js

console.log("A");
console.log("B");
```

🟦 NOTE 6: Execution vs Rendering

- Execution → synchronous
- Rendering → asynchronous
- Browser decides when to paint logs

🟩 CODE 6: Rendering delay example

```
js

for (let i = 0; i < 3; i++) {
  console.log(i);
}
```

🟦 NOTE 7: Performance cost of `console.log`

- Each call crosses **JS → host boundary**

- Formatting + DevTools rendering is expensive
- Logging inside loops is a performance issue

🟩 CODE 7: Bad performance pattern

```
js

for (let i = 0; i < 100000; i++) {
  console.log(i);
}
```

🟦 NOTE 8: Object logging uses REFERENCES

- Objects are logged by reference
- DevTools evaluates object **lazily**
- Logged value may change later

🟩 CODE 8: Reference behavior

```
js

let obj = { count: 1 };
console.log(obj);

obj.count = 100;
```

🟦 NOTE 9: Snapshot logging (safe logging)

- Spread → shallow copy
- JSON stringify → deep snapshot

🟩 CODE 9: Snapshot examples

```
js

console.log({ ...obj });
console.log(JSON.parse(JSON.stringify(obj)));
```

🟦 NOTE 10: Common console methods

- `log` → normal output
- `warn` → warnings
- `error` → errors / stderr
- `info` → informational

🟩 CODE 10: Console methods

```
js
```

```
console.log("log");
console.warn("warn");
console.error("error");
console.info("info");
```

🟦 NOTE 11: `console.table`

- DevTools-only formatting
- Useful for arrays and objects
- Not part of ECMAScript

🟩 CODE 11: Table output

```
js

console.table([
 { name: "Anoop", age: 22 },
 { name: "John", age: 25 }
]);
```

🟦 NOTE 12: Measuring time

- `console.time` starts timer
- `console.timeEnd` ends timer
- Used for quick performance checks

🟩 CODE 12: Timing code

```
js

console.time("loop");

for (let i = 0; i < 100000; i++) {}

console.timeEnd("loop");
```

🟦 NOTE 13: `console.trace`

- Prints current call stack
- Does not throw error
- Useful for debugging flow

🟩 CODE 13: Trace example

```
js
```

```
function a() {
  b();
}

function b() {
  console.trace("Stack trace");
}

a();
```

🟦 NOTE 14: `console.clear`

- Clears DevTools UI only
- Does NOT clear memory or variables

🟩 CODE 14: Clear console

```
js

console.clear();
```

🟦 NOTE 15: Console can be overridden

- Methods can be reassigned
- Often used to disable logs in production

🟩 CODE 15: Override example

```
js

console.log = function () {};
console.log("This will not print");
```

🟦 NOTE 16: Interview facts (important)

- `console` is runtime-dependent
- Not guaranteed by ECMAScript
- JS can work without console
- Logging objects logs references

🟦 FINAL MENTAL MODEL

- `console` = runtime-provided object
- Not JavaScript language feature
- Non-blocking
- Reference-based logging
- Expensive if abused

🛑 STOP HERE

Next **NotebookLM-ready** topic (same format):

1️⃣ JavaScript **Data Types**

2️⃣ `prompt / alert / confirm`

3️⃣ Execution Context (deep, code-driven)

Tell me the **number**.