Perfect. I'll now **rewrite Object Destructuring** in the **same refined, clean, deep-but-simple PDF-style** you want — with **better flow, internals, edge cases, and interview clarity**.

📘 Object Destructuring in JavaScript — Deep but Simple (PDF-Style)

🔹 1. What is Object Destructuring? (Core Idea)

Definition

Object destructuring is a syntax that allows you to **extract object properties into variables** in a single statement.

👉 You pull values **out of an object**
👉 And store them in **individual variables**

One-Line Meaning

> Object destructuring = **unpacking object properties into variables**

🔎 Internal Perspective

- No new object is created
- JavaScript performs **property lookup**
- Retrieved values are **assigned to variables**
- If value is an object → reference is assigned

✅ Key Takeaway

Destructuring is **syntax sugar**, not a new data structure.

⚠️ Interview Trap

Destructuring does **not clone objects**.

🔹 2. Basic Object Destructuring

Syntax

```js
const { prop1, prop2 } = obj;
```

Example

```js
const watch = {
  brand: "Titan",
  price: 6000
};

const { brand, price } = watch;
```

🔎 Internal Behavior

- JS looks for keys `"brand"` and `"price"`
- Assigns their values to variables with same names

Equivalent to:

```js

 const brand = watch.brand;
const price = watch.price;
```

✅ Key Takeaway

Variable names must match **object keys**.

⚠️ Interview Trap

Missing properties become `undefined`.

🔹 3. Destructuring Only Required Properties

```js

 const watch = {
  brand: "Titan",
  price: 6000,
  color: "Pink",
  dial: "Round"
};

const { brand, price } = watch;
```

🔎 Internal Behavior

- Unused properties are ignored
- Object remains unchanged
- No performance penalty

✅ Key Takeaway

You can extract **only what you need**.

🔹 4. Renaming Variables While Destructuring

Why Needed

- Avoid name conflicts
- Improve readability

Syntax

```js
const { prop: newName } = obj;
```

Example

```js
const watch = {
  brand: "Titan",
  color: "Pink",
  dial: "Round"
};

const { brand: bd, color: cr, dial: dl } = watch;
```

🔎 Internal Behavior

- Property lookup still uses original key
- New variable name is just an alias

```js
bd; // "Titan"
```

✅ Key Takeaway

Renaming changes **variable name**, not property name.

⚠️ Interview Trap

`bd` is NOT a key in the object.

🔹 5. Default Values in Object Destructuring

Why Defaults Are Needed

- Property may not exist
- Property may be `undefined`

Example

```js
```

```
const animal = {
  name: "Lion",
  age: 10
};

const { name = "Tiger", color = "Yellow" } = animal;
```

🔎 Internal Behavior

- Default applies **only if value is** `undefined`
- Default does NOT apply if value is `null`

```js
const obj = { x: null };
const { x = 10 } = obj;

x; // null
```

✅ Key Takeaway

Defaults trigger only for `undefined`.

⚠️ Interview Trap

`null` bypasses default values.

🔹 6. Renaming + Default Together (COMMON CASE)

```js
const animal = {
  name: "Lion"
};

const {
  name: animalName = "Tiger",
  color: animalColor = "Yellow"
} = animal;
```

🔎 Internal Behavior

- Lookup → rename → apply default (if needed)
- Happens left to right

✅ Key Takeaway

Renaming and defaults can be combined safely.

🔹 7. Object Destructuring with Rest Operator

Purpose

Collect remaining properties into a new object.

Syntax

```js
const { prop1, ...rest } = obj;
```

Example

```js
const nums = {
  num1: 10,
  num2: 20,
  num3: 30,
  num4: 40
};

const { num1, ...numbers } = nums;
```

🔎 Internal Behavior

- `num1` extracted first
- Remaining properties copied into new object
- Shallow copy only

```js
numbers; // { num2: 20, num3: 30, num4: 40 }
```

✅ Key Takeaway

Rest creates a **new object** with remaining properties.

⚠️ Interview Trap

Rest does NOT deep copy nested objects.

🔹 8. Object Destructuring in Function Parameters

Why Used

- Cleaner function signatures
- Avoid repetitive property access

Example

```js
function sum({ num1, num2, num3, num4 }) {
  return num1 + num2 + num3 + num4;
}

const nums = {
  num1: 5,
  num2: 7,
  num3: 10,
  num4: 12
};

sum(nums); // 34
```

🔎 Internal Behavior

- Destructuring happens at **function entry**
- Same rules apply as normal destructuring

⚠️ Danger Case

```js
sum(); // ❌ TypeError
```

Safe Version

```js
function sum({ num1, num2 } = {}) {}
```

✅ Key Takeaway

Always default destructured parameters.

🔹 9. Common Edge Cases & Traps

❌ Accessing undeclared variable

```js
const { a } = {};
a; // undefined (valid)
```

❌ Using wrong key

```js
const { brand } = { Brand: "Titan" };
brand; // undefined
```

❌ Assuming deep copy

```js
const obj = { a: { x: 1 } };
const { a } = obj;

a.x = 99;
obj.a.x; // 99
```

🧠 FINAL SUMMARY (EXAM + INTERVIEW)

- Object destructuring unpacks properties
- Variable names must match keys (unless renamed)
- Defaults apply only for `undefined`
- Rest collects remaining properties
- No deep copy happens
- Function parameter destructuring is powerful but risky
- Destructuring improves readability & maintainability

⏭️ NEXT (Choose One)

Reply with **one word**:

- `array-destructuring`
- `nested-destructuring`
- `destructuring-edge-cases`
- `spread-vs-destructuring`
- `real-world-destructuring`

I'll continue in **exact same refined PDF-style**.