Excellent. I'll now **rewrite Polymorphism** in the **same refined, clean, deep-but-simple PDF-style** as the previous three (Encapsulation, Inheritance, Abstraction).

This version will fix **common misconceptions**, explain **runtime dispatch internally**, and make it **interview-grade**.

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

🔹 1. What is Polymorphism? (Core Idea)

Definition

Polymorphism means **one interface, many forms.**

👉 Same method name

👉 Different behavior

👉 Decided at **runtime**

Word Breakdown

- **Poly** → many
- **Morph** → forms

One-Line Meaning

> Polymorphism allows different objects to respond **differently to the same method call**.

🔎 Internal Perspective

- JavaScript supports **runtime polymorphism**
- Achieved via:
  - prototype chain
  - method overriding
- JS does **NOT** support compile-time polymorphism

✅ Key Takeaway

Polymorphism = dynamic method selection.

⚠️ Interview Trap

JavaScript does NOT support method overloading like Java/C++.

🔹 2. Polymorphism vs Overloading vs Overriding

Important Distinction

Concept	JavaScript Support
Method Overloading	❌ Not supported
Method Overriding	✅ Supported
Runtime Polymorphism	✅ Supported

🔎 Why Overloading Doesn't Exist in JS

```js
function add(a, b) {}
function add(a, b, c) {} // overwrites previous
```

- JS functions are objects
- Last definition wins
- No signature-based dispatch

✅ Key Takeaway

Polymorphism in JS is achieved via **overriding**, not overloading.

🔹 3. What is Method Overriding?

Definition

Method overriding occurs when:

- Parent and child have methods with **same name**
- Child provides its **own implementation**

Rule

> Child method **overrides** parent method.

🔹 4. Polymorphism via Method Overriding (Classic Example)

Parent Class

```js
class Shape {
  area() {
    return "Area depends on shape";
  }
}
```

Child Classes

```js
class Circle extends Shape {
  area(radius) {
    return 3.14 * radius * radius;
  }
}

class Rectangle extends Shape {
  area(length, width) {
    return length * width;
  }
}
```

Usage

```js
const c = new Circle();
const r = new Rectangle();

c.area(5);    // 78.5
r.area(5, 10); // 50
```

🔎 Internal Behavior (VERY IMPORTANT)

When `c.area()` is called:

1. JS looks for `area` in `Circle.prototype`
2. Found → executes it
3. Parent method is ignored

```javascript
c
 ↓
Circle.prototype (area ✔️)
 ↓
Shape.prototype
```

✅ Key Takeaway

Method resolution is based on **object's actual type**, not reference.

⚠️ **Interview Trap**

Method lookup happens at **runtime**, not compile time.

🔹 5. Runtime Polymorphism (Core Mechanism)

Why It’s Called Runtime Polymorphism

- JS decides **which method to execute at runtime**
- Decision depends on:
  - object instance
  - prototype chain

```js
function printArea(shape) {
 console.log(shape.area());
}
```

```js
printArea(new Circle());
printArea(new Rectangle());
```

🔎 **Internal Insight**

- Same function call
- Different objects
- Different behavior

✅ **Key Takeaway**

Polymorphism enables **dynamic behavior switching**.

🔹 6. Using `super` in Overridden Methods

Purpose

Extend parent behavior instead of replacing it.

Example

```js
class MathOps {
 calculate(a, b) {
   console.log("Add:", a + b);
  }
}
```

```
class AdvancedMath extends MathOps {
  calculate(a, b) {
    super.calculate(a, b);
    console.log("Multiply:", a * b);
  }
}
```

Usage

```js
const m = new AdvancedMath();
m.calculate(10, 5);
```

🔎 Internal Behavior

- `super.calculate()` looks in parent prototype
- Executes parent method first
- Then executes child logic

✅ Key Takeaway

`super` allows **behavior extension**, not duplication.

⚠️ Interview Trap

`super` refers to **parent prototype**, not parent object.

🔹 7. Polymorphism Without Classes (Prototype-Based)

```js
const animal = {
  speak() {
    return "Animal sound";
  }
};

const dog = Object.create(animal);
dog.speak = function () {
  return "Bark";
};

dog.speak(); // "Bark"
```

🔎 Internal Behavior

- JS checks `dog` first
- Finds overridden method
- Parent method remains untouched

✅ **Key Takeaway**

Polymorphism exists **even without classes**.

🔹 8. Why JavaScript Has No Compile-Time Polymorphism

Reasons

- No function signatures
- No type enforcement
- No method overloading

```js
function test(a) {}
function test(a, b) {} // replaces previous
```

✅ **Key Takeaway**

JavaScript is **dynamically typed**, so polymorphism is runtime-only.

🔹 9. Benefits of Polymorphism

✔️ Code Reusability

Reuse parent logic via `super`

✔️ Extensibility

Add new child classes without changing existing code

✔️ Dynamic Behavior

Same interface, different outcomes

🔹 10. Common Interview Traps

- ❌ Polymorphism ≠ method overloading in JS
- ❌ Method selection is NOT compile-time
- ✔️ Overriding works via prototype chain
- ✔️ `super` works only in classes
- ✔️ Private methods cannot be overridden

🧠 FINAL SUMMARY (EXAM + INTERVIEW)

- Polymorphism = one interface, many behaviors
- JavaScript supports runtime polymorphism

- Achieved using method overriding
- Method resolution is dynamic
- Prototype chain decides execution
- No compile-time polymorphism in JS
- Enables flexible, extensible designs

⏭️ NEXT (Choose One)

Reply with **one word**:

- `oop-complete-summary`
- `js-oop-vs-java`
- `real-world-oop-design`
- `design-patterns-intro`

I'll continue with the **same clarity & depth**.