Encapsulation, inheritance, and polymorphism

Let's dive into **Encapsulation**, **Inheritance**, and **Polymorphism** — the three core principles of **Object-Oriented Programming (OOP)**, especially in the context of **JavaScript**.

📦 1. Encapsulation

Encapsulation means **bundling data and the methods that operate on that data into a single unit** (object/class) and **restricting direct access** to some components.

✅ Benefits:

- Protects internal state
- Prevents unintended interference
- Improves modularity and maintenance

🛠️ JavaScript Example (ES6 class with private fields):

```
class BankAccount {
  #balance = 0; // private field

  constructor(owner) {
    this.owner = owner;
  }

  deposit(amount) {
    if (amount > 0) this.#balance += amount;
  }

  getBalance() {
```

```
    return this.#balance;
  }
}

const account = new BankAccount("Abhi");
account.deposit(1000);
console.log(account.getBalance()); // ✅ 1000
console.log(account.#balance);     // ❌ Error: Private field
```

🧬 2. Inheritance

Inheritance allows one class (child/subclass) to **inherit properties and methods from another class** (parent/superclass), enabling **code reuse** and logical hierarchies.

✅ Benefits:

- DRY (Don't Repeat Yourself)
- Logical modeling of real-world relationships

🛠️ JavaScript Example:

```
class Animal {
  constructor(name) {
    this.name = name;
  }

  speak() {
    console.log(`${this.name} makes a noise.`);
  }
}

class Dog extends Animal {
  speak() {
    console.log(`${this.name} barks.`);
```

```
  }
}

const dog = new Dog("Buddy");
dog.speak(); // Buddy barks.
```

🌀 3. Polymorphism

Polymorphism means "**many forms**". In OOP, it refers to the ability of different classes to be **treated as instances of the same base class**, but with **behavior specific to their types**.

> Achieved via method overriding in JavaScript.

🛠️ JavaScript Example:

```
class Shape {
  area() {
    return 0;
  }
}

class Circle extends Shape {
  constructor(radius) {
    super();
    this.radius = radius;
  }

  area() {
    return Math.PI * this.radius ** 2;
  }
}

class Square extends Shape {
  constructor(side) {
```

```
    super();
    this.side = side;
  }

  area() {
    return this.side ** 2;
  }
}

const shapes = [new Circle(2), new Square(3)];

shapes.forEach(shape => {
  console.log(shape.area()); // Circle and Square compute differently
});
```

Here, both `Circle` and `Square` **override** the `area()` method from `Shape`, but are treated as `Shape` instances — **polymorphism in action**.

🧩 Summary Table

Principle	Description	Example
Encapsulation	Hides internal state, exposes behavior via methods	Private fields in classes
Inheritance	Reuse functionality from a parent class	`class Dog extends Animal`
Polymorphism	Different classes define same method differently	`area()` in `Circle`, `Square`

If you'd like to see how these work in **functional programming style** or in **React class components**, I can show those too!