OBJECT ORIENTED PROGRAMMING (OOPS) :-

1. What is OOP?

OOP is a programming paradigm based on the concept of **objects**, which contain **data (properties)** and **methods (functions)**.

2. Key Principles of OOP :-

Principle	Description
Encapsulation	Wrapping data and methods into a single unit (class), and restricting direct access.
Abstraction	Hiding complex details and showing only the essential features.
Inheritance	One class (child) inherits the properties and methods of another (parent).
Polymorphism	Same method behaves differently depending on the object calling it.

3. Basic Terminology

1. Class – Blueprint for creating objects.

2. Object – Instance of a class.

3. Constructor – Method used to initialize an object.

4. this keyword – Refers to the current instance of the class.

4. Example in JavaScript

javascript

```javascript
class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }

  greet() {
    console.log(`Hello, I’m ${this.name}`);
  }
}

const p1 = new Person("Prabhat", 25);
p1.greet(); // Hello, I’m Prabhat
```

5. Inheritance Example

```javascript
class Employee extends Person {
  constructor(name, age, jobTitle) {
    super(name, age); // Call parent constructor
    this.jobTitle = jobTitle;
  }

  work() {
    console.log(`${this.name} is working as a ${this.jobTitle}`);
  }
}
```

6. Encapsulation (with #private fields)

```javascript
class BankAccount {
  #balance = 0;

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

  getBalance() {
    return this.#balance;
  }
}
```

7. Polymorphism Example

```javascript
class Animal {
  speak() {
    console.log("Animal speaks");
  }
}

class Dog extends Animal {
  speak() {
    console.log("Dog barks");
  }
}

const a = new Dog();
a.speak(); // Dog barks
```

🔹 1. Encapsulation

Encapsulation means **bundling** the data (variables) and the methods (functions) that operate on that data into a single unit—**a class**. It also helps in **restricting access** to certain parts of an object to prevent unwanted interference.

✅ Benefits:

- Protects internal state
- Prevents direct access
- Makes code modular and easier to manage

✅ **Example:**

```javascript
class User {
  #password; // private field

  constructor(name, password) {
    this.name = name;
    this.#password = password;
  }

  checkPassword(input) {
    return input === this.#password;
  }
}

const user1 = new User("Prabhat", "secret123");

console.log(user1.name); // Prabhat
console.log(user1.#password); // ❌ Error: private field
console.log(user1.checkPassword("secret123")); // true
```

🔹 2. Abstraction

Abstraction means **hiding the internal implementation** and showing only the **necessary details**. It's like using a mobile phone—you don't need to know how it works internally to use it.

✅ **Benefits:**

- Reduces complexity
- Focus on what an object does, not how

✅ **Example:**

```javascript
class Car {
  startEngine() {
    console.log("Starting engine...");
    this.#injectFuel();
    this.#igniteSpark();
  }

  #injectFuel() {
    console.log("Fuel injected");
  }

  #igniteSpark() {
    console.log("Spark ignited");
  }
}
```

```
const myCar = new Car();
myCar.startEngine();
// Output:
// Starting engine...
// Fuel injected
// Spark ignited

myCar.#injectFuel(); // ❌ Error: can't access private method
```

🔹 3. Inheritance

Inheritance allows a class (**child/subclass**) to inherit **properties and methods** from another class (**parent/superclass**). This promotes **code reuse**.

✅ Benefits:

- Reduces redundancy
- Promotes reusability

javascript

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

  makeSound() {
    console.log("Some generic sound");
  }
}

class Dog extends Animal {
  makeSound() {
    console.log("Bark!");
  }
}

const dog = new Dog("Tommy");
dog.makeSound(); // Bark!
console.log(dog.name); // Tommy
```

🔹 4. Polymorphism

Polymorphism means "**many forms**". It allows the same method name to behave **differently based on the object that is calling it**.

✅ Benefits:

- Flexibility
- Extensibility

```javascript
class Shape {
  draw() {
    console.log("Drawing a shape");
  }
}

class Circle extends Shape {
  draw() {
    console.log("Drawing a circle");
  }
}

class Square extends Shape {
  draw() {
    console.log("Drawing a square");
  }
}

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

shapes.forEach(shape => shape.draw());
/* Output:
Drawing a circle
Drawing a square
Drawing a shape
*/
```

◆ Summary Table

OOP Concept	Purpose	JS Keyword / Usage
Encapsulation	Protect and bundle data/methods	`class`, `#private`, methods
Abstraction	Hide complexity	`#private`, helper methods
Inheritance	Reuse logic from parent classes	`extends`, `super()`
Polymorphism	Multiple behaviors for the same method call	Method overriding

5. Composition

Composition is a design principle where a class is composed of one or more objects from other classes, rather than inheriting from them. It follows the concept of:

> **"Has-a" relationship** rather than "is-a".

Instead of saying a `Car` **is a** `Engine`, we say a `Car` **has an** `Engine`.

✅ Why use Composition?

- More **flexible** than inheritance
- Encourages **modular and reusable code**
- Avoids deep inheritance trees (which can get messy)

✅ JavaScript Example of Composition:

```javascript
class Engine {
  start() {
    console.log("Engine started");
  }
}

class Wheels {
  rotate() {
    console.log("Wheels are rotating");
  }
}

class Car {
  constructor() {
    this.engine = new Engine();
    this.wheels = new Wheels();
  }

  drive() {
    this.engine.start();
    this.wheels.rotate();
    console.log("Car is driving");
  }
}

const myCar = new Car();
myCar.drive();
/* Output:
Engine started
Wheels are rotating
Car is driving
*/
```

📌 Instead of extending `Engine`, the `Car` **uses** `Engine` and `Wheels`—this is **composition**.

🔹 6. Method Overriding

Method Overriding means a **child class** provides a **specific implementation** of a method that is already defined in its **parent class**.

This is a key part of **polymorphism**.

✅ JavaScript Example:

```javascript
class Animal {
  speak() {
    console.log("Animal makes a sound");
  }
}

class Cat extends Animal {
  speak() {
    console.log("Cat meows");
  }
}

const a = new Animal();
const c = new Cat();

a.speak(); // Animal makes a sound
c.speak(); // Cat meows
```

Here, `Cat` **overrides** the `speak()` method from `Animal`.

✅ When to use Method Overriding:

- When a subclass needs to **customize** or **completely change** the behavior of an inherited method.
- To implement **specific behavior** while keeping a common interface.

🔸 Inheritance vs Composition (Quick Comparison)

Feature	Inheritance	Composition
Relationship	"Is-a" (Dog is an Animal)	"Has-a" (Car has an Engine)

Flexibility	Less flexible, tightly coupled	More flexible, loosely coupled
Reusability	Reuses via parent class	Reuses via delegation (object usage)