Java Object-Oriented Programming (OOP) – Complete Guide with Comments

This file is a **full beginner-to-strong-intermediate guide** to Java OOP. Every code snippet now includes detailed **comments explaining each line**.

---

1. Why Object-Oriented Programming Exists

Before OOP, programs were written as long lists of instructions (procedural programming). As software grew, code became: - Hard to maintain - Hard to reuse - Easy to break

OOP solves this by modeling software the same way we think about the real world — using objects.

An object: - Has **state** (data) - Has **behavior** (actions)

---

2. Classes and Objects

Example

```
// Define a class called Student
class Student {
    String name; // Field to store the student's name
    int age;     // Field to store the student's age

    // Method representing behavior
    void study() {
        System.out.println(name + " is
studying"); // Prints a message including the student's name
    }
}

// Main method to run code
public class Main {
    public static void main(String[] args) {
        Student s1 = new Student(); // Create a new Student object
        s1.name = "Alex";          // Set the student's name
        s1.age = 20;                // Set the student's age
        s1.study();                 // Call the study method
    }
}
```

---

3. Fields (Attributes)

```
class Car {
    String brand; // The brand of the car
    int speed;    // Current speed of the car
}
```

---

4. Methods

```
class Car {
    int speed;

    // Method to accelerate the car
    void accelerate() {
        speed += 10; // Increase speed by 10
        System.out.println("Current speed: " + speed);
    }
}
```

---

5. Constructors

```
class User {
    String username;

    // Constructor initializes the username
    User(String username) {
        this.username = username; // 'this' refers to the current object's field
    }
}
```

---

6. The `this` Keyword

```
class Person {
    String name;

    // Constructor with parameter name
```

```
    Person(String name) {
        this.name = name; // Assign parameter to the object's field
    }
}
```

---

7. Access Modifiers

```
class BankAccount {
    private double balance; // Only accessible inside this class

    // Public getter allows controlled access
    public double getBalance() {
        return balance;
    }
}
```

---

8. Encapsulation

```
class BankAccount {
    private double balance; // Hidden from outside

    // Method to safely deposit money
    public void deposit(double amount) {
        if (amount > 0) {
            balance += amount; // Only valid deposits affect balance
        }
    }
}
```

---

9. Inheritance

```
class Animal {
    void eat() {
        System.out.println("Eating...");
    }
}

class Dog extends Animal { // Dog inherits from Animal
```

```
    void bark() {
        System.out.println("Barking...");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.eat();  // Inherited method
        dog.bark(); // Dog's own method
    }
}
```

---

10. Method Overriding

```
class Animal {
    void sound() {
        System.out.println("Animal sound");
    }
}

class Cat extends Animal {
    @Override
    void sound() { // Override parent method
        System.out.println("Meow");
    }
}
```

---

11. Polymorphism

```
Animal a = new Dog(); // Reference type Animal, actual object is Dog
// Calls Dog's methods at runtime if overridden
```

---

12. Method Overloading

```
class MathUtil {
    int add(int a, int b) { // Two-parameter version
        return a + b;
```

```
    }

    int add(int a, int b, int c) { // Three-parameter version
        return a + b + c;
    }
}
```

---

13. Abstraction

```
// Abstract class defines method without implementation
abstract class Shape {
    abstract double area(); // Must be implemented by subclass
}

// Interface defines a contract
interface Payment {
    void pay(double amount); // Any implementing class must provide pay method
}
```

---

14. Static Keyword

```
class Utils {
    static int add(int a, int b) { // Belongs to class, not object
        return a + b;
    }
}
```

---

15. Final Keyword

```
final class Constants { // Cannot be inherited
    static final double PI = 3.14159; // Constant value
}
```

---

16. Composition vs Inheritance

```
class Engine {
    void start() {}
}

class Car {
    Engine engine = new Engine(); // Car has an Engine
}
```

---

17. Packages

```
package com.app.models; // Organizes classes into a namespace
```

---

18. Common Beginner Mistakes

- Making all fields public
- Overusing inheritance
- Ignoring constructors
- Not modeling real-world behavior

---

19. Final Practice Assignment

Design a **Library System** with: - Book - Member - Librarian

Use: - Encapsulation - Inheritance - Polymorphism - Interfaces

Try adding comments explaining each line as practice.