Java Object-Oriented Programming (OOP) Cheat Sheet

Four Pillars of OOP

1. Encapsulation

2. Inheritance

3. Polymorphism

4. Abstraction

---

1. ENCAPSULATION

Definition: Bundling data (fields) and methods that operate on that data within a single unit (class), and restricting direct access to some components.

Basic Encapsulation

```java
```

```java
public class Person {
    // Private fields - hidden from outside
    private String name;
    private int age;
    private String email;

    // Constructor
    public Person(String name, int age, String email) {
        this.name = name;
        this.age = age;
        this.email = email;
    }

    // Getter methods - read access
    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

    public String getEmail() {
        return email;
    }

    // Setter methods - write access with validation
    public void setName(String name) {
        if (name != null && !name.isEmpty()) {
            this.name = name;
        }
    }

    public void setAge(int age) {
        if (age > 0 && age < 150) {
            this.age = age;
        }
    }

    public void setEmail(String email) {
        if (email.contains("@")) {
            this.email = email;
        }
    }
}
```

```java
// Usage
Person person = new Person("John", 25, "john@email.com");
System.out.println(person.getName());  // Accessing via getter
person.setAge(26);  // Modifying via setter
```

Benefits of Encapsulation

- **Data Hiding**: Protects object state from unauthorized access
- **Validation**: Control how data is set and accessed
- **Flexibility**: Change internal implementation without affecting external code
- **Maintainability**: Easier to modify and debug

Encapsulation with Read-Only Fields

java

```
public class Book {
    private final String isbn;  // Immutable
    private String title;
    private double price;

    public Book(String isbn, String title, double price) {
        this.isbn = isbn;
        this.title = title;
        this.price = price;
    }

    // Only getter for isbn (read-only)
    public String getIsbn() {
        return isbn;
    }

    // Both getter and setter for title
    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    // Getter with business logic
    public double getPrice() {
        return price;
    }

    public void setPrice(double price) {
        if (price >= 0) {
            this.price = price;
        } else {
            throw new IllegalArgumentException("Price cannot be negative");
        }
    }

    // Calculated/derived property
    public double getPriceWithTax() {
        return price * 1.18;  // 18% tax
    }
}
```

2. INHERITANCE

Definition: Mechanism where a new class (child/subclass) acquires properties and behaviors of an existing class (parent/superclass).

Basic Inheritance

java

```java
// Parent class (Superclass)
public class Animal {
    protected String name;
    protected int age;

    public Animal(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public void eat() {
        System.out.println(name + " is eating");
    }

    public void sleep() {
        System.out.println(name + " is sleeping");
    }

    public void makeSound() {
        System.out.println("Some generic sound");
    }
}

// Child class (Subclass)
public class Dog extends Animal {
    private String breed;

    // Constructor calling parent constructor
    public Dog(String name, int age, String breed) {
        super(name, age);  // Call parent constructor
        this.breed = breed;
    }

    // Override parent method
    @Override
    public void makeSound() {
        System.out.println(name + " says: Woof! Woof!");
    }

    // New method specific to Dog
    public void fetch() {
        System.out.println(name + " is fetching the ball");
    }

    public String getBreed() {
        return breed;
    }
}

// Another child class
public class Cat extends Animal {
    private boolean isIndoor;

    public Cat(String name, int age, boolean isIndoor) {
        super(name, age);
        this.isIndoor = isIndoor;
    }

    @Override
    public void makeSound() {
        System.out.println(name + " says: Meow!");
    }

    public void scratch() {
        System.out.println(name + " is scratching");
    }
}

// Usage
Dog dog = new Dog("Buddy", 3, "Golden Retriever");
dog.eat();        // Inherited from Animal
dog.makeSound();  // Overridden in Dog
dog.fetch();      // Specific to Dog

Cat cat = new Cat("Whiskers", 2, true);
cat.sleep();      // Inherited from Animal
cat.makeSound();  // Overridden in Cat
cat.scratch();    // Specific to Cat
```

Inheritance Hierarchy

java

```java
// Grandparent class
public class LivingBeing {
    public void breathe() {
        System.out.println("Breathing...");
    }
}

// Parent class
public class Animal extends LivingBeing {
    public void move() {
        System.out.println("Moving...");
    }
}

// Child class
public class Mammal extends Animal {
    public void feedMilk() {
        System.out.println("Feeding milk to young");
    }
}

// Grandchild class
public class Human extends Mammal {
    public void speak() {
        System.out.println("Speaking...");
    }
}

// Usage - Human has access to all methods
Human human = new Human();
human.breathe();   // From LivingBeing
human.move();      // From Animal
human.feedMilk();  // From Mammal
human.speak();     // From Human
```

Types of Inheritance in Java

java

```
// Single Inheritance
class A { }
class B extends A { }

// Multilevel Inheritance
class A { }
class B extends A { }
class C extends B { }

// Hierarchical Inheritance
class A { }
class B extends A { }
class C extends A { }

// NOTE: Multiple Inheritance is NOT supported with classes
// class C extends A, B { }  // ERROR!
// Use interfaces for multiple inheritance
```

Using `super` Keyword

java

```
public class Vehicle {
    protected String brand;
    protected int year;

    public Vehicle(String brand, int year) {
        this.brand = brand;
        this.year = year;
    }

    public void displayInfo() {
        System.out.println("Brand: " + brand);
        System.out.println("Year: " + year);
    }
}

public class Car extends Vehicle {
    private int numberOfDoors;

    public Car(String brand, int year, int doors) {
        super(brand, year);  // Call parent constructor
        this.numberOfDoors = doors;
    }

    @Override
    public void displayInfo() {
        super.displayInfo();  // Call parent method
        System.out.println("Doors: " + numberOfDoors);
    }

    public void showBrand() {
        System.out.println(super.brand);  // Access parent field
    }
}
```

Method Overriding Rules

java

```java
public class Parent {
    // Cannot be overridden
    public final void finalMethod() { }

    // Can be overridden
    public void normalMethod() { }

    // Static method - hiding, not overriding
    public static void staticMethod() { }
}

public class Child extends Parent {
    // Correct override
    @Override
    public void normalMethod() {
        // New implementation
    }

    // This is method hiding, not overriding
    public static void staticMethod() {
        // Different implementation
    }

    // ERROR: Cannot override final method
    // public void finalMethod() { }
}
```

---

3. POLYMORPHISM

Definition: Ability of objects to take multiple forms. Same interface, different implementations.

Types of Polymorphism

A. Compile-Time Polymorphism (Method Overloading)

java

```
public class Calculator {
    // Same method name, different parameters

    public int add(int a, int b) {
        return a + b;
    }

    public int add(int a, int b, int c) {
        return a + b + c;
    }

    public double add(double a, double b) {
        return a + b;
    }

    public String add(String a, String b) {
        return a + b;
    }
}

// Usage
Calculator calc = new Calculator();
System.out.println(calc.add(5, 3));           // 8
System.out.println(calc.add(5, 3, 2));        // 10
System.out.println(calc.add(5.5, 3.2));       // 8.7
System.out.println(calc.add("Hello", "World")); // HelloWorld
```

B. Runtime Polymorphism (Method Overriding)

```
java
```

```java
public class Shape {
    public void draw() {
        System.out.println("Drawing a shape");
    }

    public double calculateArea() {
        return 0;
    }
}

public class Circle extends Shape {
    private double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    @Override
    public void draw() {
        System.out.println("Drawing a circle");
    }

    @Override
    public double calculateArea() {
        return Math.PI * radius * radius;
    }
}

public class Rectangle extends Shape {
    private double length;
    private double width;

    public Rectangle(double length, double width) {
        this.length = length;
        this.width = width;
    }

    @Override
    public void draw() {
        System.out.println("Drawing a rectangle");
    }

    @Override
    public double calculateArea() {
        return length * width;
    }
}

// Usage - Runtime Polymorphism
Shape shape1 = new Circle(5);
Shape shape2 = new Rectangle(4, 6);

shape1.draw();  // Drawing a circle
shape2.draw();  // Drawing a rectangle

System.out.println(shape1.calculateArea());  // Circle area
System.out.println(shape2.calculateArea());  // Rectangle area

// Polymorphic array
Shape[] shapes = {
    new Circle(3),
    new Rectangle(5, 7),
    new Circle(4)
};

for (Shape shape : shapes) {
    shape.draw();
    System.out.println("Area: " + shape.calculateArea());
}
```

Upcasting and Downcasting

java

```
// Upcasting (Implicit)
Animal animal = new Dog("Max", 5, "Labrador");  // Dog -> Animal
animal.eat();        // Works
animal.makeSound();  // Works
// animal.fetch();   // ERROR: Animal doesn't have fetch()

// Downcasting (Explicit)
if (animal instanceof Dog) {
    Dog dog = (Dog) animal;  // Cast to Dog
    dog.fetch();  // Now works
}

// Using instanceof for safety
public void feedAnimal(Animal animal) {
    animal.eat();

    if (animal instanceof Dog) {
        Dog dog = (Dog) animal;
        dog.fetch();
    } else if (animal instanceof Cat) {
        Cat cat = (Cat) animal;
        cat.scratch();
    }
}
```

Constructor Overloading

java

```java
public class Student {
    private String name;
    private int age;
    private String major;

    // No-arg constructor
    public Student() {
        this.name = "Unknown";
        this.age = 0;
        this.major = "Undeclared";
    }

    // Constructor with name
    public Student(String name) {
        this.name = name;
        this.age = 0;
        this.major = "Undeclared";
    }

    // Constructor with name and age
    public Student(String name, int age) {
        this.name = name;
        this.age = age;
        this.major = "Undeclared";
    }

    // Constructor with all parameters
    public Student(String name, int age, String major) {
        this.name = name;
        this.age = age;
        this.major = major;
    }
}

// Better approach using constructor chaining
public class Student {
    private String name;
    private int age;
    private String major;

    public Student() {
        this("Unknown", 0, "Undeclared");
    }

    public Student(String name) {
        this(name, 0, "Undeclared");
    }

    public Student(String name, int age) {
        this(name, age, "Undeclared");
    }

    public Student(String name, int age, String major) {
        this.name = name;
        this.age = age;
        this.major = major;
    }
}
```

4. ABSTRACTION

Definition: Hiding complex implementation details and showing only essential features.

Abstract Classes

java

```
public abstract class BankAccount {
    protected String accountNumber;
    protected double balance;

    public BankAccount(String accountNumber, double balance) {
        this.accountNumber = accountNumber;
        this.balance = balance;
    }

    // Abstract methods
    public abstract void withdraw(double amount);
    public abstract double calculateInterest();

    // Concrete method
    public void deposit(double amount) {
        balance += amount;
        System.out.println("Deposited: " + amount);
    }

    public double getBalance() {
        return balance;
    }
}

public class SavingsAccount extends BankAccount {
    private double interestRate;

    public SavingsAccount(String accountNumber, double balance, double rate) {
        super(accountNumber, balance);
        this.interestRate = rate;
    }

    @Override
    public void withdraw(double amount) {
        if (balance >= amount + 100) {  // Minimum balance
            balance -= amount;
            System.out.println("Withdrawn: " + amount);
        } else {
            System.out.println("Insufficient balance");
        }
    }

    @Override
    public double calculateInterest() {
        return balance * interestRate / 100;
    }
}
```

Interfaces

java

```java
public interface Drawable {
    // Abstract method (implicitly public abstract)
    void draw();

    // Default method (Java 8+)
    default void display() {
        System.out.println("Displaying drawable object");
    }

    // Static method (Java 8+)
    static void info() {
        System.out.println("This is a drawable interface");
    }

    // Constant (implicitly public static final)
    int MAX_SIZE = 100;
}

public interface Resizable {
    void resize(int width, int height);
}

// Implementing single interface
public class Circle implements Drawable {
    private int radius;

    @Override
    public void draw() {
        System.out.println("Drawing circle with radius: " + radius);
    }
}

// Implementing multiple interfaces
public class Rectangle implements Drawable, Resizable {
    private int width;
    private int height;

    @Override
    public void draw() {
        System.out.println("Drawing rectangle: " + width + "x" + height);
    }

    @Override
    public void resize(int width, int height) {
        this.width = width;
        this.height = height;
    }
}
```

Abstract Class vs Interface

java

```
// Abstract Class Example
public abstract class Vehicle {
    protected String brand;  // Can have fields

    public Vehicle(String brand) {  // Can have constructor
        this.brand = brand;
    }

    public abstract void start();  // Abstract method

    public void stop() {  // Concrete method
        System.out.println("Vehicle stopped");
    }
}

// Interface Example
public interface Drivable {
    // No fields (only constants)
    int MAX_SPEED = 200;

    // No constructor

    void accelerate();  // Abstract method
    void brake();       // Abstract method

    default void honk() {  // Default method
        System.out.println("Beep beep!");
    }
}

// Class can extend one abstract class and implement multiple interfaces
public class Car extends Vehicle implements Drivable {
    public Car(String brand) {
        super(brand);
    }

    @Override
    public void start() {
        System.out.println(brand + " car started");
    }

    @Override
    public void accelerate() {
        System.out.println("Car accelerating");
    }
    @Override
    public void brake() {
        System.out.println("Car braking");
    }
}
```

ADDITIONAL OOP CONCEPTS

Static Members

java

```java
public class Counter {
    // Static variable - shared by all instances
    private static int count = 0;

    // Instance variable
    private int id;

    // Constructor
    public Counter() {
        count++;
        this.id = count;
    }

    // Static method - can access only static members
    public static int getCount() {
        return count;
    }

    // Instance method
    public int getId() {
        return id;
    }

    // Static block - executes once when class is loaded
    static {
        System.out.println("Counter class loaded");
        count = 0;
    }
}

// Usage
Counter c1 = new Counter();
Counter c2 = new Counter();
Counter c3 = new Counter();

System.out.println(Counter.getCount());  // 3
System.out.println(c1.getId());  // 1
System.out.println(c2.getId());  // 2
```

Inner Classes

java

```
public class OuterClass {
    private String outerField = "Outer";

    // Inner class
    public class InnerClass {
        public void display() {
            System.out.println("Accessing: " + outerField);
        }
    }

    // Static nested class
    public static class StaticNestedClass {
        public void show() {
            System.out.println("Static nested class");
        }
    }

    // Method local inner class
    public void method() {
        class LocalInnerClass {
            public void print() {
                System.out.println("Local inner class");
            }
        }

        LocalInnerClass local = new LocalInnerClass();
        local.print();
    }
}

// Usage
OuterClass outer = new OuterClass();
OuterClass.InnerClass inner = outer.new InnerClass();
inner.display();

OuterClass.StaticNestedClass nested = new OuterClass.StaticNestedClass();
nested.show();
```

Anonymous Classes

java

```
// Interface
interface Greeting {
    void greet();
}

// Anonymous class implementation
Greeting greeting = new Greeting() {
    @Override
    public void greet() {
        System.out.println("Hello from anonymous class!");
    }
};

greeting.greet();

// Anonymous class extending a class
Thread thread = new Thread() {
    @Override
    public void run() {
        System.out.println("Running in thread");
    }
};

thread.start();
```

Object Class Methods

java

```
public class Product {
    private String name;
    private double price;

    public Product(String name, double price) {
        this.name = name;
        this.price = price;
    }

    // Override toString()
    @Override
    public String toString() {
        return "Product{name='" + name + "', price=" + price + "}";
    }

    // Override equals()
    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null || getClass() != obj.getClass()) return false;
        Product product = (Product) obj;
        return Double.compare(product.price, price) == 0
            && name.equals(product.name);
    }

    // Override hashCode()
    @Override
    public int hashCode() {
        int result = name.hashCode();
        result = 31 * result + Double.hashCode(price);
        return result;
    }
}
```

COMPLETE OOP EXAMPLE

```
java
```

```java
// Interface
interface Payable {
    double calculateSalary();
    void displayPaymentInfo();
}

// Abstract class
abstract class Employee implements Payable {
    protected String name;
    protected int id;
    protected String department;

    public Employee(String name, int id, String department) {
        this.name = name;
        this.id = id;
        this.department = department;
    }

    // Concrete method
    public void displayInfo() {
        System.out.println("ID: " + id);
        System.out.println("Name: " + name);
        System.out.println("Department: " + department);
    }

    // Abstract method
    public abstract void work();

    @Override
    public void displayPaymentInfo() {
        System.out.println("Salary: $" + calculateSalary());
    }
}

// Concrete class 1
class FullTimeEmployee extends Employee {
    private double monthlySalary;
    private double bonus;

    public FullTimeEmployee(String name, int id, String dept,
                           double salary, double bonus) {
        super(name, id, dept);
        this.monthlySalary = salary;
        this.bonus = bonus;
    }
    @Override
    public void work() {
        System.out.println(name + " is working full-time");
    }

    @Override
    public double calculateSalary() {
        return monthlySalary + bonus;
    }
}

// Concrete class 2
class PartTimeEmployee extends Employee {
    private int hoursWorked;
    private double hourlyRate;

    public PartTimeEmployee(String name, int id, String dept,
                           int hours, double rate) {
        super(name, id, dept);
        this.hoursWorked = hours;
        this.hourlyRate = rate;
    }

    @Override
    public void work() {
        System.out.println(name + " is working part-time");
    }

    @Override
    public double calculateSalary() {
        return hoursWorked * hourlyRate;
    }
}

// Usage demonstrating all OOP concepts
public class Main {
    public static void main(String[] args) {
        // Polymorphism - treating different objects uniformly
        Employee[] employees = {
            new FullTimeEmployee("Alice", 101, "IT", 5000, 1000),
            new PartTimeEmployee("Bob", 102, "HR", 80, 25),
            new FullTimeEmployee("Charlie", 103, "Finance", 6000, 1500)
        };

        for (Employee emp : employees) {
            emp.displayInfo();
            emp.work();
            emp.displayPaymentInfo();
            System.out.println();
        }
    }
}
```

OOP BEST PRACTICES

1. Favor Composition Over Inheritance

java

```java
// Instead of inheritance
class Car extends Engine { }  // BAD

// Use composition
class Car {
    private Engine engine;  // GOOD

    public Car(Engine engine) {
        this.engine = engine;
    }
}
```

2. Program to Interface

java

```java
// Instead of concrete class
ArrayList<String> list = new ArrayList<>();  // Works

// Program to interface
List<String> list = new ArrayList<>();  // Better
```

3. Follow SOLID Principles

- **S**ingle Responsibility: One class, one purpose
- **O**pen/Closed: Open for extension, closed for modification
- **L**iskov Substitution: Subtypes must be substitutable for base types
- **I**nterface Segregation: Many specific interfaces better than one general
- **D**ependency Inversion: Depend on abstractions, not concrete classes

4. Use Access Modifiers Properly

java

```
private   // Most restrictive - use by default
protected // For inheritance hierarchies
public    // For public API only
```

5. Override toString(), equals(), and hashCode()

java

```
// Always override these for custom objects
@Override
public String toString() { }

@Override
public boolean equals(Object obj) { }

@Override
public int hashCode() { }
```

QUICK REFERENCE

Concept	Keyword	Purpose
Inheritance	`extends`	Inherit from class
Interface Implementation	`implements`	Implement interface
Abstract Class	`abstract`	Cannot be instantiated
Abstract Method	`abstract`	Must be overridden
Method Override	`@Override`	Override parent method
Final Class	`final`	Cannot be extended
Final Method	`final`	Cannot be overridden
Final Variable	`final`	Cannot be changed
Static Member	`static`	Belongs to class
Super	`super`	Access parent
This	`this`	Reference current object