**Section 6: Java OOP Notes**

**1. Methods**

**Key Concepts**:

* Blocks of code performing specific tasks.
* Improve code reusability and organization.
* Types: User-defined and standard library methods (e.g., String.length()).

**Implementation**:

java

*// Declaration*

static void sayHello() { *// void = no return*

System.out.println("Hello");

}

*// Method with return*

static int addNumbers(int x, int y) {

return x + y;

}

*// Method with parameters*

static int addNumbers(int x, int y) {

return x + y;

}

*// Calling methods*

sayHello(); *// Output: "Hello"*

int sum = addNumbers(10, 20); *// Returns 30*

**Tools/APIs**: Java Standard Library (e.g., Math.sqrt()).  
**Best Practices**:

* Use descriptive method names.
* Overload methods for similar functionality with different parameters.
* Always include return for non-void methods.

**2. Classes and Objects**

**Key Concepts**:

* **Class**: Blueprint defining states (fields) and behaviors (methods).
* **Object**: Instance of a class (e.g., Car redCar = new Car()).

**Implementation**:

java

*// Class definition*

public class Car {

int speed; *// Field (state)*

int year;

void accelerate() { *// Method (behavior)*

speed += 10;

}

}

*// Creating an object*

Car redCar = new Car();

redCar.speed = 180;

redCar.accelerate();

**Best Practices**:

* Group related fields/methods in a class.
* Use nouns for class names (e.g., Car, User).

**3. Constructors**

**Key Concepts**:

* Special methods initializing objects during creation.
* Same name as the class.

**Implementation**:

java

public class Car {

int speed;

int year;

*// Constructor*

public Car(int carSpeed, int carYear) {

speed = carSpeed;

year = carYear;

}

}

*// Usage*

Car redCar = new Car(100, 2023); *// Sets speed=100, year=2023*

**Best Practices**:

* Initialize all critical fields in constructors.
* Overload constructors for flexibility.

**4. Access Modifiers**

**Key Concepts**:  
Control visibility of classes/methods/fields:

* public: Accessible everywhere.
* private: Accessible only within the same class.
* protected: Accessible in same package + subclasses.
* Default: Accessible only in same package.

**Implementation**:

java

public class Car {

public int speed; *// Accessible anywhere*

private int year; *// Accessible only in Car*

protected String model; *// Accessible in subclasses*

}

**Best Practices**:

* Encapsulate fields as private (use getters/setters).
* Use public for APIs, private for internal logic.

**5. Encapsulation**

**Key Concepts**:

* Protect data by restricting direct access.
* Use getters/setters to interact with private fields.

**Implementation**:

java

public class Car {

private int year;

*// Getter*

public int getYear() {

return year;

}

*// Setter*

public void setYear(int newYear) {

year = newYear;

}

}

*// Usage*

redCar.setYear(2023);

int year = redCar.getYear();

**Tools**: Android Studio generates getters/setters (Alt + Insert).  
**Best Practices**:

* Validate data in setters (e.g., if (newYear > 2000)).

**6. Inheritance**

**Key Concepts**:

* Subclass inherits fields/methods from superclass.
* Promotes code reuse.
* Keyword: extends.

**Implementation**:

java

*// Superclass*

public class Vehicle {

int numberOfWheels;

void start() { System.out.println("Vehicle starts"); }

}

*// Subclass*

public class Car extends Vehicle {

String model;

void honk() { System.out.println("Beep!"); }

}

*// Usage*

Car myCar = new Car();

myCar.start(); *// Inherited from Vehicle*

**Best Practices**:

* Use for "is-a" relationships (e.g., Car **is a** Vehicle).
* Avoid deep inheritance hierarchies.

**7. Method Overriding**

**Key Concepts**:

* Subclass redefines a method from its superclass.
* Same method signature (name, return type, parameters).

**Implementation**:

java

public class Vehicle {

void start() { System.out.println("Vehicle starts"); }

}

public class Car extends Vehicle {

@Override *// Annotation (optional but recommended)*

void start() { System.out.println("Car starts"); }

}

*// Usage*

Car myCar = new Car();

myCar.start(); *// Output: "Car starts"*

**Best Practices**:

* Use @Override for compiler checks.
* Call superclass method with super.start() if needed.

**8. Polymorphism**

**Key Concepts**:

* Treat objects of subclasses as superclass type.
* Enables code flexibility.

**Implementation**:

java

Vehicle v1 = new Car(); *// Polymorphic assignment*

Vehicle v2 = new Truck();

v1.start(); *// Calls Car's start()*

v2.start(); *// Calls Truck's start()*

**Best Practices**:

* Design methods around supertypes (e.g., void repair(Vehicle v)).

**9. Abstraction**

**Key Concepts**:

* Hide implementation details, expose functionality.
* **Abstract Classes**: Cannot be instantiated; may contain abstract/non-abstract methods.
* **Interfaces**: Define contracts (method signatures) for classes to implement.

**Implementation (Abstract Class)**:

java

public abstract class Vehicle {

abstract void brake(); *// No implementation*

}

public class Car extends Vehicle {

@Override

void brake() { System.out.println("Mechanical brake"); }

}

**Implementation (Interface)**:

java

public interface VehicleInterface {

void startEngine();

}

public class Car implements VehicleInterface {

@Override

public void startEngine() { */\* Implementation \*/* }

}

**Best Practices**:

* Use interfaces for multiple inheritance.
* Abstract classes for shared code with partial implementations.

**10. Exercise: Shape Area Calculation**

**Problem**: Calculate area/perimeter for shapes (Circle, Rectangle, etc.).  
**Solution with Abstraction**:

java

public interface Shape {

double calculateArea();

double calculatePerimeter();

}

public class Circle implements Shape {

private double radius;

public Circle(double r) { radius = r; }

@Override

public double calculateArea() {

return 3.14 \* radius \* radius;

}

}

public class Rectangle implements Shape {

private double length, width;

public Rectangle(double l, double w) {

length = l; width = w;

}

@Override

public double calculateArea() {

return length \* width;

}

}

**Best Practices**:

* Force implementation via interfaces.
* Add new shapes without modifying existing code.

**Part B: Important Topics Not Covered**

**1. Composition vs. Inheritance**

* **Composition**: Build complex objects by combining simpler ones (e.g., Car has an Engine).
* **Why**: Avoids fragile base class problem; more flexible than inheritance.
* **Industry Shift**: Prefer composition ("has-a") over inheritance ("is-a").

**2. SOLID Principles**

* **Single Responsibility**: A class should have one reason to change.
* **Open/Closed**: Classes open for extension, closed for modification.
* **Liskov Substitution**: Subclasses should replace superclasses without breaking code.
* **Interface Segregation**: Avoid bulky interfaces; split into smaller ones.
* **Dependency Inversion**: Depend on abstractions, not concretions.

**3.**static**Keyword**

* **Purpose**: Shared class-level variables/methods (not per-instance).
* **Use Cases**: Constants, utility methods (e.g., Math.max()).
* **Example**:

java

public class Car {

static int count = 0; *// Shared across all cars*

public Car() { count++; }

}

**4. Enums**

* **Purpose**: Define fixed set of constants (e.g., days of the week).
* **Example**:

java

public enum Day { MON, TUE, WED }

Day today = Day.MON;

**5. Generics**

* **Purpose**: Type-safe classes/collections (e.g., ArrayList<String>).
* **Example**:

java

public class Box<T> {

private T content;

public void set(T item) { content = item; }

}

Box<String> stringBox = new Box<>();

**6. Exception Handling**

* **Keywords**: try, catch, finally, throw, throws.
* **Best Practice**: Catch specific exceptions; avoid empty catch blocks.

**7. Immutable Objects**

* **Principle**: Objects whose state cannot change after creation.
* **How**: Declare fields final; no setters.
* **Benefit**: Thread safety, simpler code.

**8. Java Memory Management**

* **Stack vs. Heap**: Local variables (stack), objects (heap).
* **Garbage Collection**: Automatic memory reclamation; avoid memory leaks.

**9. Design Patterns in Android**

* **Singleton**: Single instance of a class (e.g., database helper).
* **Observer**: Event handling (e.g., LiveData in Android).
* **Factory**: Create objects without exposing instantiation logic.

**10. Modern Java Features (Post-Java 8)**

* **Lambdas**: Concise functional interfaces.

java

Runnable r = () -> System.out.println("Hello");

* **Streams**: Process sequences of elements (e.g., filtering, mapping).
* Optional: Handle null values explicitly.

**11. Dependency Injection (Dagger/Hilt)**

* **Purpose**: Decouple components; easier testing.
* **Android Standard**: Hilt (simplified Dagger) for MVVM/MVI architectures.

**12. Concurrency**

* **Threads**: Runnable, Thread, ExecutorService.
* **Async in Android**: AsyncTask (deprecated), Coroutines (modern).

**Summary**:  
Section 6 covered foundational Java OOP concepts. For professional Android development, prioritize **composition**, **SOLID principles**, **dependency injection**, and **coroutines**. Always encapsulate fields, use interfaces for flexibility, and avoid deep inheritance hierarchies.

**Section 6: Java OOP Notes**

**1. Methods**

**Key Concepts**:

* Code blocks performing specific tasks
* Improve reusability and organization
* Types: User-defined (e.g., drawCircle()) and standard library (e.g., Math.sqrt())

**Implementation**:

java

*// Void method (no return)*

public static void sayHello() {

System.out.println("Hello from method");

}

*// Method with return value*

public static int addNumbers(int x, int y) {

return x + y; *// Must match return type*

}

*// Method overloading (same name, different params)*

public static double addNumbers(double a, double b) {

return a + b;

}

*// Calling methods*

public static void main(String[] args) {

sayHello();

int sum = addNumbers(10, 20); *// Calls int version*

double total = addNumbers(5.5, 4.5); *// Calls double version*

}

**Best Practices**:

* Use descriptive names (calculateArea() > calc())
* Validate parameters in methods
* Overload instead of creating similar-named methods
* **Android Studio Tip**: Alt + Enter to fix errors

**2. Classes & Objects**

**Key Concepts**:

* Class: Blueprint (states + behaviors)
* Object: Instance of class (e.g., Car myCar = new Car())

**Implementation**:

java

public class Car {

*// Fields (state)*

int speed;

String model;

*// Method (behavior)*

void accelerate() {

speed += 10;

}

}

*// Create object*

Car redCar = new Car();

redCar.speed = 100;

redCar.accelerate();

**Best Practices**:

* Use nouns for class names (User, Database)
* Group related fields/methods
* Initialize fields in constructors

**3. Constructors**

**Key Concepts**:

* Special method for object initialization
* Same name as class

**Implementation**:

java

public class Car {

int speed;

int year;

*// Constructor*

public Car(int speed, int year) {

this.speed = speed; *// 'this' resolves naming conflicts*

this.year = year;

}

}

*// Usage*

Car tesla = new Car(120, 2023);

**Best Practices**:

* Initialize critical fields in constructors
* Overload constructors for flexibility
* Use this keyword when parameter names match field names

**4. Access Modifiers**

**Visibility Control**:

| **Modifier** | **Class** | **Package** | **Subclass** | **World** |
| --- | --- | --- | --- | --- |
| public | ✅ | ✅ | ✅ | ✅ |
| protected | ✅ | ✅ | ✅ | ❌ |
| default | ✅ | ✅ | ❌ | ❌ |
| private | ✅ | ❌ | ❌ | ❌ |

**Best Practices**:

* Fields: private (use getters/setters)
* Public APIs: public
* Related classes: Same package + default

**5. Encapsulation**

**Key Concepts**:

* Protect data via private fields + public methods

**Implementation**:

java

public class BankAccount {

private double balance; *// Private field*

*// Getter*

public double getBalance() {

return balance;

}

*// Setter with validation*

public void deposit(double amount) {

if(amount > 0) balance += amount;

}

}

*// Usage*

BankAccount acc = new BankAccount();

acc.deposit(500);

double money = acc.getBalance();

**Best Practices**:

* Validate data in setters
* Use Android Studio: Alt + Insert → Generate Getters/Setters
* Make fields final for immutability where possible

**6. Inheritance**

**Key Concepts**:

* extends keyword
* "Is-a" relationship (e.g., Car is a Vehicle)

**Implementation**:

java

*// Superclass*

public class Vehicle {

int wheels;

void start() {

System.out.println("Engine started");

}

}

*// Subclass*

public class Car extends Vehicle {

String model;

void honk() {

System.out.println("Beep!");

}

}

*// Usage*

Car sedan = new Car();

sedan.start(); *// Inherited method*

**Best Practices**:

* Avoid deep inheritance chains (>3 levels)
* Use final to prevent inheritance where needed
* Prefer composition over inheritance

**7. Method Overriding**

**Key Concepts**:

* Subclass redefines superclass method
* Same signature (name + parameters)

**Implementation**:

java

public class Vehicle {

void start() {

System.out.println("Generic engine start");

}

}

public class Tesla extends Vehicle {

@Override *// Recommended annotation*

void start() {

System.out.println("Silent electric start");

}

}

**Best Practices**:

* Always use @Override annotation
* Call superclass method with super.methodName()
* Cannot override static/final methods

**8. Polymorphism**

**Key Concepts**:

* "Many forms" - treat objects as superclass type

**Implementation**:

java

Vehicle v1 = new Car(); *// Car treated as Vehicle*

Vehicle v2 = new Truck(); *// Truck treated as Vehicle*

v1.start(); *// Calls Car's start()*

v2.start(); *// Calls Truck's start()*

**Best Practices**:

* Design methods using supertypes: void repair(Vehicle v)
* Enables flexible code extensions

**9. Abstraction**

**Abstract Classes**:

java

public abstract class Shape {

*// Abstract method (no implementation)*

public abstract double area();

*// Concrete method*

public void display() {

System.out.println("Shape type");

}

}

**Interfaces**:

java

public interface Drivable {

void accelerate();

void brake();

}

public class Car implements Drivable {

@Override

public void accelerate() { */\* implementation \*/* }

}

**Best Practices**:

* Use interfaces for multiple inheritance
* Abstract classes for shared partial implementations
* Default methods in interfaces (Java 8+)

**10. Shape Exercise Solution**

**Problem**: Calculate area for multiple shapes

**Solution with Interface**:

java

public interface Shape {

double calculateArea();

double calculatePerimeter();

}

public class Circle implements Shape {

private double radius;

public Circle(double r) { radius = r; }

@Override

public double calculateArea() {

return Math.PI \* radius \* radius;

}

}

public class Rectangle implements Shape {

private double width, height;

public Rectangle(double w, double h) {

width = w;

height = h;

}

@Override

public double calculateArea() {

return width \* height;

}

}

*// Usage*

Shape circle = new Circle(5);

Shape rect = new Rectangle(4, 6);

System.out.println("Circle area: " + circle.calculateArea());

**Part B: Essential Topics Not Covered**

1. **Composition over Inheritance**
   * Prefer Car has Engine over Car extends Engine
   * Reduces fragile base class problem
2. **SOLID Principles**
   * **S**ingle Responsibility: 1 class = 1 purpose
   * **O**pen/Closed: Extendable without modification
   * **L**iskov Substitution: Substitutable base types
   * **I**nterface Segregation: Small focused interfaces
   * **D**ependency Inversion: Depend on abstractions
3. static**Keyword**

java

public class Counter {

static int count = 0; *// Class-level variable*

public Counter() { count++; }

}

1. **Enums**

java

public enum TrafficLight { RED, YELLOW, GREEN }

TrafficLight light = TrafficLight.RED;

1. **Exception Handling**

java

try {

int result = 10 / 0;

} catch (ArithmeticException e) {

System.out.println("Cannot divide by zero!");

} finally {

*// Cleanup code*

}

1. **Modern Java Features**
   * Lambdas: Runnable r = () -> System.out.println("Hello");
   * Streams: list.stream().filter(x -> x>5).collect(Collectors.toList())
   * Optional: Safely handle null values
2. **Android-Specific**
   * **Dependency Injection**: Hilt for MVVM architectures
   * **Concurrency**: Use coroutines instead of AsyncTask (deprecated)
   * **Lifecycle Awareness**: LiveData in ViewModel

**Critical Best Practices**:

1. Always encapsulate fields with private + getters/setters
2. Use interfaces for cross-component communication
3. Validate parameters in public methods
4. Prefer composition for complex object relationships
5. Use @Override annotation for clear overrides
6. Make immutable objects (final fields) where possible

"OOP isn't about inheritance hierarchies - it's about creating manageable, decoupled components through encapsulation and precise interfaces." - Course Insight