

OOP, Multithreading & Collections Notes

Master Object-Oriented Programming principles, Multithreading concepts, and Collections framework with comprehensive examples and best practices.

Encapsulation

Inheritance

Polymorphism

Thread Safety

Collections

Design Patterns

Concurrency

Object-Oriented Programming Fundamentals

OOP is a programming paradigm based on the concept of "objects" which contain data and methods. The four main principles are:

Encapsulation

Bundling data with methods that operate on that data, and restricting direct access to some components.

Inheritance

Creating new classes from existing ones, inheriting attributes and methods while adding new features.

Polymorphism

Objects of different types can be accessed through the same interface, with each type providing its own implementation.

Abstraction

Hiding complex implementation details and showing only essential features to the user.

OOP Implementation Example

Java OOP Example

Copy

```
// Abstract class demonstrating Abstraction
abstract class Vehicle {
    private String brand; // Encapsulation: private field
    private String model;
    private int year;

    // Constructor
    public Vehicle(String brand, String model, int year) {
        this.brand = brand;
        this.model = model;
        this.year = year;
    }
}
```

Quick Navigation

OOP Fundamentals

 SOLID Principles

 Multithreading

 Collections

 Interview Qs

 Best Practices

Code Examples

OOP Examples

Complete OOP implementation with all principles

Threading Examples

Multithreading patterns & synchronization

Collection Examples

All collection types with benchmarks

Hot Topics

 Virtual Threads (Java 21)

 Stream API

🔒 Immutable Collections

🧩 Design Patterns

🎛️ Performance Tuning

📌 Quick Quiz

? Which is faster?

ArrayList get() vs LinkedList get()

Answer: ArrayList O(1) vs LinkedList O(n)

💡 Memory Question

Where are static variables stored?

Answer: Method Area (Perm Gen/Metaspace)

★ TKTips.org Strategy

Offer this as a premium PDF guide for Java interview preparation. Bundle with mock interviews and coding exercises.

```
// Getter and Setter methods for Encapsulation
public String getBrand() {
    return brand;
}

public void setBrand(String brand) {
    this.brand = brand;
}

// Abstract method - implementation will be provided by subclasses
public abstract void start();

// Concrete method
public void displayInfo() {
    System.out.println("Brand: " + brand + ", Model: " + model + ", Year: " + year);
}

// Inheritance: Car inherits from Vehicle
class Car extends Vehicle {
    private int numberOfDoors;

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

    // Polymorphism: Overriding abstract method
    @Override
    public void start() {
        System.out.println("Car is starting with key ignition");
    }

    // Method overloading (Compile-time polymorphism)
    public void start(String keyType) {
        System.out.println("Car is starting with " + keyType + " key");
    }
}

// Inheritance: ElectricCar inherits from Car
class ElectricCar extends Car {
    private int batteryCapacity;

    public ElectricCar(String brand, String model, int year, int doors, int batteryCapacity) {
        super(brand, model, year, doors);
        this.batteryCapacity = batteryCapacity;
    }

    // Polymorphism: Overriding parent method
    @Override
    public void start() {
        System.out.println("Electric car is starting silently");
    }

    // New method specific to ElectricCar
    public void chargeBattery() {
        System.out.println("Charging battery...");
    }
}
```

```
// Interface demonstrating Polymorphism
interface Maintainable {
    void performMaintenance();
    int getServiceInterval();
}

// Multiple inheritance through interfaces
class ServiceCar extends Car implements Maintainable {
    public ServiceCar(String brand, String model, int year, int doors) {
        super(brand, model, year, doors);
    }

    @Override
    public void performMaintenance() {
        System.out.println("Performing maintenance on service car");
    }

    @Override
    public int getServiceInterval() {
        return 6; // months
    }
}

// Main class to demonstrate OOP concepts
public class Main {
    public static void main(String[] args) {
        // Polymorphism in action
        Vehicle myCar = new Car("Toyota", "Camry", 2023, 4);
        Vehicle myElectricCar = new ElectricCar("Tesla", "Model 3", 2023, 4, 75);

        // Runtime polymorphism - different start() implementations
        myCar.start(); // Calls Car's start()
        myElectricCar.start(); // Calls ElectricCar's start()

        // Compile-time polymorphism - method overloading
        Car car = new Car("Honda", "Civic", 2022, 4);
        car.start("remote"); // Calls overloaded start() method

        // Interface usage
        Maintainable serviceVehicle = new ServiceCar("Ford", "Transit", 2021, 5);
        serviceVehicle.performMaintenance();
    }
}
```

💡 **Key Insight:** Always program to interfaces, not implementations. This makes your code more flexible and maintainable.

🧩 SOLID Design Principles

SOLID principles are five design principles that make software designs more understandable, flexible, and maintainable.

S Single Responsibility Principle (SRP)

Basic

A class should have only one reason to change, meaning it should have only one job.

Before SRP

```
class User {  
    void saveToDatabase() { ... }  
    void sendEmail() { ... }  
    void generateReport() { ... }  
}
```

After SRP

```
class User { ... }  
class UserRepository { void save(User user) { ... } }  
class EmailService { void sendEmail(User user) { ... } }  
class ReportGenerator { void generate(User user) { ... } }
```

O Open/Closed Principle (OCP)

Intermediate

Software entities should be open for extension but closed for modification.

```
interface Discount {  
    double apply(double price);  
}  
  
class PercentageDiscount implements Discount {  
    private double percentage;  
    @Override public double apply(double price) {  
        return price * (1 - percentage / 100);  
    }  
}  
  
class FixedAmountDiscount implements Discount {  
    private double amount;  
    @Override public double apply(double price) {  
        return Math.max(0, price - amount);  
    }  
}
```

L Liskov Substitution Principle (LSP)

Advanced

Objects of a superclass should be replaceable with objects of its subclasses without breaking the application.

Violation Example: Square extending Rectangle where `setWidth()` and `setHeight()` behave differently.

I Interface Segregation Principle (ISP)

Intermediate

Clients should not be forced to depend on interfaces they do not use. Create specific interfaces instead of one general-purpose interface.

D Dependency Inversion Principle (DIP)

Advanced

High-level modules should not depend on low-level modules. Both should depend on abstractions.

```
// Instead of:
class ReportGenerator {
    private MySQLDatabase database; // Direct dependency
}

// Use:
class ReportGenerator {
    private Database database; // Dependency on abstraction
}
```

⚡ Multithreading & Concurrency

Multithreading allows concurrent execution of two or more threads to maximize CPU utilization.

Thread States

NEW → RUNNABLE
RUNNABLE → RUNNING
RUNNING → WAITING
WAITING → RUNNABLE
RUNNING → TERMINATED
RUNNING → BLOCKED

Thread Creation Methods

</> Extending Thread Class

```
class MyThread extends Thread {
    public void run() {
        System.out.println("Thread running");
    }
}

// Usage:
MyThread t1 = new MyThread();
t1.start();
```

⚙️ Implementing Runnable

```
class MyRunnable implements Runnable {
    public void run() {
        System.out.println("Runnable running");
    }
}

// Usage:
Thread t2 = new Thread(new MyRunnable());
t2.start();
```

Thread Synchronization

Synchronization Example

[Copy](#)

```
class Counter {
    private int count = 0;

    // Synchronized method
    public synchronized void increment() {
        count++;
    }

    // Synchronized block
    public void decrement() {
        synchronized (this) {
            count--;
        }
    }

    public synchronized int getCount() {
        return count;
    }
}

class CounterThread extends Thread {
    private Counter counter;

    public CounterThread(Counter counter) {
        this.counter = counter;
    }

    @Override
    public void run() {
        for (int i = 0; i < 1000; i++) {
            counter.increment();
        }
    }
}

public class ThreadSyncDemo {
    public static void main(String[] args) throws InterruptedException {
        Counter counter = new Counter();

        // Create multiple threads
        Thread t1 = new CounterThread(counter);
        Thread t2 = new CounterThread(counter);
        Thread t3 = new CounterThread(counter);

        // Start all threads
        t1.start();
        t2.start();
        t3.start();

        // Wait for all threads to complete
        t1.join();
        t2.join();
        t3.join();

        System.out.println("Final count: " + counter.getCount()); // Should be 3000
    }
}
```

```
}  
}  
}
```

Concurrency Utilities (java.util.concurrent)

Utility	Purpose	Example
ExecutorService	Thread pool management	Executors.newFixedThreadPool(10)
CountDownLatch	Wait for multiple threads to complete	new CountDownLatch(3)
CyclicBarrier	Threads wait at barrier point	new CyclicBarrier(3)
Semaphore	Control access to resources	new Semaphore(5)
ConcurrentHashMap	Thread-safe HashMap	new ConcurrentHashMap<>()
BlockingQueue	Thread-safe queue	new ArrayBlockingQueue<>(100)

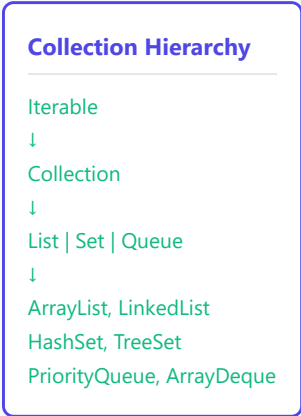
ExecutorService ExampleCopy

```
import java.util.concurrent.*;  
  
public class ExecutorServiceDemo {  
    public static void main(String[] args) {  
        // Create thread pool with 4 threads  
        ExecutorService executor = Executors.newFixedThreadPool(4);  
  
        // Submit tasks  
        for (int i = 1; i <= 10; i++) {  
            int taskId = i;  
            executor.submit() -> {  
                System.out.println("Task " + taskId + " executed by " +  
                    Thread.currentThread().getName());  
                try {  
                    Thread.sleep(1000); // Simulate work  
                } catch (InterruptedException e) {  
                    Thread.currentThread().interrupt();  
                }  
            }  
        };  
    }  
  
    // Shutdown executor  
    executor.shutdown();  
  
    try {  
        // Wait for all tasks to complete  
        if (!executor.awaitTermination(60, TimeUnit.SECONDS)) {  
            executor.shutdownNow();  
        }  
    } catch (InterruptedException e) {  
        executor.shutdownNow();  
        Thread.currentThread().interrupt();  
    }  
}
```

```
}  
}  
}
```

Collections Framework

The Java Collections Framework provides a set of interfaces and classes for storing and manipulating groups of data.



List Interface Implementations

List Type	When to Use	Time Complexity	Thread Safe?
ArrayList	Frequent read operations, random access	get: O(1), add: O(1) amortized	No
LinkedList	Frequent insertions/deletions at ends	add/remove: O(1), get: O(n)	No
Vector	Legacy, thread-safe alternative	Similar to ArrayList	Yes
CopyOnWriteArrayList	Frequent reads, rare writes	Read: O(1), Write: O(n)	Yes

Set Interface Implementations

HashSet

- Uses hash table
- No ordering guaranteed
- O(1) for add, remove, contains

🌲 TreeSet

- Uses Red-Black tree
- Sorted order (natural/comparator)
- O(log n) for operations

- Allows one null element

- No null elements

LinkedHashSet

- Hash table + linked list
- Maintains insertion order
- O(1) for operations
- Allows one null element

Map Interface Implementations

Map Examples

[Copy](#)

```
import java.util.*;

public class MapExamples {
    public static void main(String[] args) {
        // HashMap - No ordering, O(1) operations
        Map hashMap = new HashMap<>();
        hashMap.put("Alice", 25);
        hashMap.put("Bob", 30);
        hashMap.put("Charlie", 35);

        // Iterating through HashMap
        System.out.println("HashMap entries:");
        for (Map.Entry entry : hashMap.entrySet()) {
            System.out.println(entry.getKey() + ": " + entry.getValue());
        }

        // TreeMap - Sorted by keys
        Map treeMap = new TreeMap<>(hashMap);
        System.out.println("\nTreeMap (sorted): " + treeMap);

        // LinkedHashMap - Maintains insertion order
        Map linkedHashMap = new LinkedHashMap<>();
        linkedHashMap.put("Zoe", 40);
        linkedHashMap.put("Alice", 25);
        linkedHashMap.put("Bob", 30);
        System.out.println("LinkedHashMap (insertion order): " + linkedHashMap);

        // ConcurrentHashMap - Thread-safe
        Map concurrentMap = new ConcurrentHashMap<>();
        concurrentMap.put("One", 1);
        concurrentMap.put("Two", 2);

        // Useful Map methods
        System.out.println("\nMap Operations:");
        System.out.println("Contains key 'Alice': " + hashMap.containsKey("Alice"));
        System.out.println("Get or default: " + hashMap.getOrDefault("David", 0));

        // Compute if absent
        hashMap.computeIfAbsent("David", k -> 40);
        System.out.println("After computeIfAbsent: " + hashMap);
    }
}
```

```
// Merge values
hashMap.merge("Alice", 5, Integer::sum);
System.out.println("After merge: " + hashMap);
}
}
```

Collection Algorithms & Best Practices

- ✓ **Choose the right collection:** ArrayList for random access, LinkedList for frequent insertions/deletions
- ✓ **Use diamond operator:** List list = new ArrayList<>();
- ✓ **Prefer for-each loop:** for (String item : list) { ... }
- ✓ **Use Collections utility class:** Collections.sort(), Collections.reverse(), Collections.shuffle()
- ✓ **Initialize with capacity:** new ArrayList<>(1000) for large lists

? Interview Questions

Q1: Difference between ArrayList and LinkedList?

ArrayList: Backed by array, fast random access ($O(1)$), slow insertions/deletions ($O(n)$ except at end).

LinkedList: Doubly linked list, slow random access ($O(n)$), fast insertions/deletions ($O(1)$ if position known).

Q2: What is the volatile keyword in Java?

Ensures visibility of changes to variables across threads. Prevents thread caching and guarantees reading/writing from/to main memory.

Q3: Difference between HashMap and ConcurrentHashMap?

HashMap: Not thread-safe, allows one null key and multiple null values.

ConcurrentHashMap: Thread-safe, doesn't allow null keys/values, uses lock striping for better concurrency.

Q4: What is the difference between Runnable and Callable?

Runnable: run() method returns void, cannot throw checked exceptions.

Callable: call() method returns a value and can throw checked exceptions.

Q5: Explain the producer-consumer problem solution.

Use BlockingQueue or wait()/notify() pattern with synchronized blocks to coordinate between producers and consumers.

★ Best Practices & Tips

🛡️ Thread Safety Best Practices

- ✓ Use synchronized collections when needed: `Collections.synchronizedList()`
- ✓ Prefer concurrent collections (`ConcurrentHashMap`, `CopyOnWriteArrayList`) over synchronized wrappers
- ✓ Use Atomic classes (`AtomicInteger`, `AtomicReference`) for simple atomic operations
- ✓ Always use `ExecutorService` instead of creating threads manually
- ✓ Use `ThreadLocal` for thread-specific variables

</> OOP Best Practices

- ✓ Follow SOLID principles religiously
- ✓ Prefer composition over inheritance
- ✓ Use immutable objects whenever possible
- ✓ Program to interfaces, not implementations
- ✓ Use dependency injection for better testability

🎓 **Pro Tip:** Master these three areas (OOP, Multithreading, Collections) thoroughly - they form the foundation for senior Java developer roles and system design interviews.

About TKTips.org Java

Comprehensive Java programming guides covering OOP, Multithreading, Collections, and advanced concepts for interview preparation.



Java Topics

Core Java
Multithreading
Collections
Design Patterns
Spring Framework

Resources

Interview Questions
Code Examples
Cheat Sheets
Practice Problems
Community Forum

Newsletter

Get weekly Java tips, interview questions, and coding challenges.

Your email address

Subscribe

© 2026 TKTips.org - OOP, Multithreading & Collections Guide. All rights reserved.

Java is a trademark of Oracle Corporation.