Complete Java Notes (Parts A–I)

TABLE OF CONTENTS

1. Part A — Core Java Foundation
2. Part B — Advanced OOP & Language Features
3. Part C — Exception Handling + Assertions
4. Part D — Multithreading & Concurrency
5. Part E — Collections & Generics
6. Part F — Modern Java Features (Java 5–17)
7. Part G — Java APIs & Tools (I/O, JDBC, Logging, etc.)
8. Part H — Best Practices & Pitfalls
9. Part I — Practice Material (Interview Qs + Coding Exercises + Mini Projects)

KEY FEATURES INCLUDED

- Theory + Definitions + Full Forms
- Diagrams (JVM, Collections, Threads, etc.)
- Practical Code Examples
- Interview Questions (Easy to Tough)
- Java versions के नए features
- Best Practices & Common Pitfalls
- Mini Projects & Exercises

🚀 INDEX

🕮 Part A — Core Java Foundation **01-05**

✓ History, Features, Applications
✓ JVM, JDK, JRE, IDE tools
✓ Data types, Variables, Operators, Unicode, Type casting
✓ Control statements, Loops, Jumping
✓ Arrays (1D, 2D, Jagged) + Utility methods
✓ Methods, Recursion, Varargs
✓ OOPs (Class, Object, Encapsulation, Abstraction, Inheritance, Polymorphism)
✓ Constructors, this, super, static, final, access modifiers

🕮 Part B — Advanced OOP & Language Features **06-10**

✓ Abstract classes & Interfaces (Java 8+ additions, Functional Interfaces)
✓ Packages, Access Control
✓ Strings & StringBuffer vs StringBuilder
✓ Wrapper classes, Autoboxing/Unboxing
✓ Enum, Annotations (built-in + custom)
✓ Memory Management (Heap, Stack, Method Area, PC register)
✓ Java keywords (volatile, transient, synchronized, native, strictfp, etc.)

🕮 Part C — Exception Handling **11-14**

✓ Exception hierarchy diagram
✓ try-catch-finally, throw vs throws, multiple catch
✓ Checked vs Unchecked vs Error
✓ Custom exceptions
✓ Best Practices

🕮 Part D — Multithreading & Concurrency **15-19**

✓ Thread creation (Thread vs Runnable)
✓ Life cycle diagram
✓ Synchronization, Locks, Deadlock, Starvation, Livelock
✓ Inter-thread communication (wait/notify)
✓ Concurrency API (ExecutorService, Future, CountDownLatch, Semaphore)
✓ Daemon threads & Thread Pool

🕮 Part E — Collections & Generics **20-25**

✓ Collection hierarchy diagram
✓ List, Set, Queue, Map (implementations & differences)
✓ Comparable vs Comparator
✓ Fail-fast vs Fail-safe iterators
✓ Generics in depth (bounded, wildcards)

🕮 Part F — Modern Java (Java 5 → 17 features) 26-30

✓ Generics, Enum, Varargs (Java 5)
✓ String in switch, try-with-resources (Java 7)
✓ Lambdas, Streams, Optional (Java 8)
✓ Modules (Java 9)
✓ var keyword, HttpClient API (Java 11)
✓ Records, Sealed classes, Pattern Matching (Java 14–17)

🕮 Part G — Java APIs & Tools 31-35

✓ File I/O (classic + NIO.2)
✓ Serialization & Deserialization
✓ Streams API (map, filter, reduce, collect)
✓ Date & Time API (java.time)
✓ JDBC (CRUD operations, PreparedStatement, Transactions)
✓ Logging API (java.util.logging, Log4j overview)

🕮 Part H — Best Practices + Pitfalls 36-40

✓ Why String is immutable?
✓ When to use equals() vs ==
✓ Why multiple inheritance avoided?
✓ Coding standards (naming, comments, design choices)
✓ Security pitfalls (SQL injection prevention with PreparedStatement, avoiding raw serialization)

🕮 Part I — Practice Material 41-44

✓ 200+ Interview Questions (basic → advanced)
✓ Short coding tasks (prime, Fibonacci, matrix ops, anagrams, deadlock simulation, file read/write, etc.)
✓ Mini projects:

- Banking System (OOP + Multithreading)
- Library Management (Collections)
- Student Record Manager (JDBC)
- Log Analyzer (File I/O + Streams)

📌 **Part A (Core Java Foundation) |**

📘 JAVA NOTES — PART A

(Core Java Foundation)

CH 1 — Java का परिचय

1.1 What is Java?

- **Java = Programming Language + Platform**
- Invented by **James Gosling (Sun Microsystems, 1995)**
- अब **Oracle Corporation** maintain करती है।
- Tagline: **WORA (Write Once, Run Anywhere)**

1.2 Java की विशेषताएँ (Features of Java)

- **Simple** → C++ जैसा syntax, लेकिन pointers/operator overloading नहीं।
- **Object-Oriented (OOP)** → Class, Object, Inheritance, Polymorphism, Abstraction, Encapsulation।
- **Platform Independent** → Java code compile होकर **Bytecode (.class file)** बनाता है। JVM उस bytecode को हर OS पर run करता है।
- **Secure** → No pointers, Bytecode Verifier, Security Manager।
- **Robust** → Automatic Garbage Collection, Strong type checking।
- **Multithreaded** → Concurrent execution।
- **Distributed** → Network-based applications possible।
- **Dynamic** → Classes run-time पर load हो सकती हैं।

1.3 Applications of Java

1. **Desktop Applications** → Media Player, Antivirus
2. **Web Applications** → IRCTC, Flipkart
3. **Enterprise Applications** → Banking, ERP
4. **Mobile Applications** → Android Apps
5. **Embedded Systems** → Smart Cards, IoT
6. **Games & Robotics**

1.4 Important Full Forms

- **JVM** → Java Virtual Machine
- **JRE** → Java Runtime Environment
- **JDK** → Java Development Kit
- **API** → Application Programming Interface
- **IDE** → Integrated Development Environment

CH 2 — Java Environment Setup

2.1 JDK, JRE, JVM

- **JVM (Java Virtual Machine):** Bytecode को execute करता है।
- **JRE (Java Runtime Environment):** JVM + Libraries (run करने के लिए)।
- **JDK (Java Development Kit):** JRE + Tools (compiler, debugger, etc.)

☞ **Shortcut**:

- Develop करने के लिए = **JDK**
- सिर्फ run करने के लिए = **JRE**
- Run internally करने वाला engine = **JVM**

2.2 Important Java Tools

- `javac` → Compiler
- `java` → Launcher
- `javadoc` → Documentation generator
- `jar` → Archive tool
- `jdb` → Debugger

2.3 First Program

```
class Hello {
   public static void main(String[] args) {
      System.out.println("Hello Java");
   }
}
```

Execution:

```
javac Hello.java   # compile
```

```
java Hello          # run
```

CH 3 — Java Syntax & Conventions

- **Identifiers** → Names (letters, digits, _, $) → Example: `sum`, `StudentName`
- **Reserved Keywords** → `class`, `public`, `static`, `if`, `else`, etc.
- **Naming Convention**
  - Class → PascalCase (e.g., `StudentInfo`)
  - Variable → camelCase (e.g., `studentName`)
  - Constant → UPPER_CASE (e.g., `MAX_VALUE`)

CH 4 — Data Types in Java

4.1 Primitive Data Types (8)

Type	Size	Example
byte	1 byte	byte b = 10;
short	2 bytes	short s = 200;
int	4 bytes	int x = 1000;
long	8 bytes	long l = 100000L;
float	4 bytes	float f = 3.14f;
double	8 bytes	double d = 123.45;
char	2 bytes	char c = 'A';
boolean	1 bit	boolean flag = true;

4.2 Non-Primitive Data Types

- String, Arrays, Classes, Interfaces

4.3 Type Casting

- **Widening (Implicit):** int → long → float → double
- **Narrowing (Explicit):** double → float → int → byte

```
int x = (int) 5.9; // narrowing
```

CH 5 — Variables

1. **Local Variable** → method के अंदर declared।
2. **Instance Variable** → class के अंदर, method के बाहर।
3. **Static Variable** → `static` keyword, सभी objects में shared।

CH 6 — Operators

- **Arithmetic** → `+ - * / % ++ --`
- **Relational** → `< <= > >= == !=`
- **Logical** → `&& || !`
- **Bitwise** → `& | ^ ~ << >> >>>`
- **Assignment** → `=, +=, -=, *=, /=`
- **Ternary** → `condition ? a : b`

CH 7 — Control Statements

7.1 if-else

```
if(x>0) { System.out.println("Positive"); }
else { System.out.println("Negative"); }
```

7.2 switch

```
switch(day) {
   case 1: System.out.println("Mon"); break;
   default: System.out.println("Other");
}
```

7.3 Loops

- for, while, do-while, for-each

7.4 Jump Statements

- `break`, `continue`, `return`

💡 **Part A में हमने**: Introduction → JVM/JDK/JRE → Syntax → Data Types → Variables → Operators → Control Statements को cover किया।

☞ **Part B (Advanced OOP & Language Features)** ।इसमें OOPs की गहराई + Java keywords + Memory concepts cover होंगे।

📚 JAVA NOTES — PART B

(Advanced OOP & Language Features)

CH 8 — Object-Oriented Programming (OOP) Basics

8.1 Class and Object

- **Class** → Blueprint/template for objects.
- **Object** → Instance of class (state + behavior).

Example:

```
class Car {
   String color;
   void drive() { System.out.println("Driving..."); }
}
class Test {
   public static void main(String[] args) {
      Car c1 = new Car();
      c1.color = "Red";
      c1.drive();
   }
}
```

8.2 Four Pillars of OOP

1. **Encapsulation** → Data hiding using `private` + getters/setters.
2. **Abstraction** → Hiding implementation, showing only functionality.
3. **Inheritance** → Code reusability (IS-A relationship).
4. **Polymorphism** → One thing, many forms (overloading/overriding).

CH 9 — Constructors

- Special method → object initialization.
- Name = Class name, no return type.

9.1 Types of Constructors

1. **Default Constructor** → provided automatically (if no constructor given).
2. **Parameterized Constructor** → accepts arguments.
3. **Constructor Overloading** → multiple constructors with different parameters.

Example:

```
class Student {
   int id;
   String name;

   Student() { id = 0; name = "Unknown"; }
   Student(int i, String n) { id = i; name = n; }
}
```

CH 10 — this, super, static, final

10.1 this keyword

- Current object का reference.
- Uses: instance variable differentiation, constructor chaining.

10.2 super keyword

- Parent class reference.
- Uses: parent constructor call, parent method access.

10.3 static keyword

- Class-level member.
- Shared by all objects.
- Can be: variable, method, block.

10.4 final keyword

- final variable → constant.
- final method → cannot override.
- final class → cannot extend.

CH 11 — Inheritance

- **Single Inheritance** → One parent, one child.
- **Multilevel Inheritance** → Grandparent → Parent → Child.
- **Hierarchical Inheritance** → One parent, many children.
- **Multiple Inheritance** → Not supported in classes (diamond problem), but via interfaces possible.

CH 12 — Polymorphism

12.1 Method Overloading (Compile-time Polymorphism)

- Same method name, different parameters.

```
class Calc {
   int add(int a, int b) { return a+b; }
   double add(double a, double b) { return a+b; }
}
```

12.2 Method Overriding (Runtime Polymorphism)

- Subclass redefines parent method.

```
class A { void show(){ System.out.println("A"); } }
class B extends A { void show(){ System.out.println("B"); } }
```

CH 13 — Abstraction

13.1 Abstract Class

- May have abstract + non-abstract methods.
- Cannot be instantiated.

```
abstract class Shape {
   abstract void draw();
}
class Circle extends Shape {
   void draw(){ System.out.println("Drawing Circle"); }
}
```

13.2 Interface

- Pure abstraction (till Java 7).
- Java 8 → default & static methods.
- Java 9 → private methods in interface.

```
interface Printable {
   void print();
   default void info(){ System.out.println("Info"); }
}
```

CH 14 — Encapsulation

- Binding of data + methods into one unit.
- Use of `private` variables + public getters/setters.

Example:

```
class Employee {
   private int salary;
   public void setSalary(int s){ salary = s; }
   public int getSalary(){ return salary; }
```

```
}
```

CH 15 — Packages & Access Modifiers

15.1 Packages

- Collection of related classes/interfaces.
- Syntax:

```
package mypack;
```

- Use: `import java.util.*;`

15.2 Access Modifiers

Modifier	Same Class	Same Package	Subclass	World
private	✓	✗	✗	✗
default	✓	✓	✗	✗
protected	✓	✓	✓	✗
public	✓	✓	✓	✓

CH 16 — Strings

16.1 String (Immutable)

```
String s1 = "Hello";   // String literal (pooled)
String s2 = new String("Hello"); // new object
```

- Immutable = once created, cannot change.

16.2 StringBuffer & StringBuilder

- **StringBuffer** → Mutable + Thread-safe.
- **StringBuilder** → Mutable + Faster (not synchronized).

CH 17 — Wrapper Classes

- Convert primitive → object (boxing) & object → primitive (unboxing).
- Classes: `Integer, Float, Double, Character, Boolean, Byte, Short, Long`.

```
int a = 10;
Integer obj = a;    // Autoboxing
int b = obj;        // Unboxing
```

CH 18 — Memory Management in Java

18.1 JVM Memory Areas

- **Heap** → Objects
- **Stack** → Method calls, local vars
- **Method Area** → Class info, static data
- **PC Register** → Current instruction
- **Native Method Stack** → Native code execution

18.2 Garbage Collection (GC)

- Automatically removes unused objects.
- `System.gc()` → request GC.
- `finalize()` → cleanup before GC (deprecated in Java 9+).

💡 Part B में हमने cover किया:

- OOP (Class, Object, Inheritance, Polymorphism, Abstraction, Encapsulation)
- Important keywords (`this`, `super`, `static`, `final`)
- Packages & Access Modifiers
- Strings, Wrapper Classes
- JVM Memory Management

👉 Part C (Exception Handling + Assertions) ।

📘 JAVA NOTES — PART C

(Exception Handling + Assertions)

CH 19 — Exception Handling

19.1 What is Exception?

- Exception = Runtime पर आने वाली **error-like condition**, जो normal program flow को disturb कर देती है।
- Java में हर exception एक **object** है, जो **Throwable class** से inherit करता है।

☞ Full Form:

- **JVM** = Java Virtual Machine
- **JRE** = Java Runtime Environment
- **JDK** = Java Development Kit

19.2 Types of Exceptions

1. **Checked Exceptions** (Compile-time exceptions)
   - Compiler आपको force करता है handle करने के लिए।
   - Example: `IOException`, `SQLException`, `ClassNotFoundException`
2. **Unchecked Exceptions** (Runtime exceptions)
   - Compiler check नहीं करता।
   - Example: `NullPointerException`, `ArithmeticException`, `ArrayIndexOutOfBoundsException`
3. **Errors**
   - Serious problems, जिन्हें program handle नहीं कर सकता।
   - Example: `OutOfMemoryError`, `StackOverflowError`

19.3 Exception Hierarchy (Diagram Explanation)

```
Throwable
│
├── Exception
│       ├── IOException
│       ├── SQLException
│       └── RuntimeException
│               ├── NullPointerException
│               ├── ArithmeticException
│               └── IndexOutOfBoundsException
│
└── Error
        ├── OutOfMemoryError
        ├── StackOverflowError
```

19.4 Exception Handling Keywords

1. **try** → Risky code block
2. **catch** → Exception handler block
3. **finally** → हमेशा execute होता है (cleanup के लिए)
4. **throw** → एक single exception को explicitly फेंकना
5. **throws** → Method declaration में exceptions specify करना

19.5 Example 1 (Basic Try-Catch)

```
public class Example1 {
   public static void main(String[] args) {
      try {
         int a = 10 / 0;
      } catch (ArithmeticException e) {
         System.out.println("Error: " + e.getMessage());
      }
   }
}
```

☞ Output: `Error: / by zero`

19.6 Example 2 (Multiple Catch + Finally)

```
try {
   int[] arr = new int[3];
   arr[5] = 10;
} catch (ArithmeticException e) {
   System.out.println("Arithmetic Error");
} catch (ArrayIndexOutOfBoundsException e) {
   System.out.println("Array Error");
} finally {
```

```
   System.out.println("Finally Block Always Runs");
}
```

19.7 throw vs throws

- **throw** → used inside method to throw exception.

```
throw new IOException("File not found");
```

- **throws** → method declaration में used.

```
void readFile() throws IOException { ... }
```

19.8 Custom Exception Example

```
class AgeException extends Exception {
   AgeException(String msg) { super(msg); }
}

public class Test {
   public static void main(String[] args) {
      try {
         int age = 15;
         if(age < 18) throw new AgeException("Not eligible to vote");
      } catch (AgeException e) {
         System.out.println(e.getMessage());
      }
   }
}
```

CH 20 — Assertions

20.1 What is Assertion?

- Assertion = Testing assumption in program.
- Syntax:

```
assert condition;
assert condition : "Error message";
```

20.2 Example

```
int age = 16;
assert age >= 18 : "Age must be 18 or above";
System.out.println("Age is " + age);
```

☞ Run with:

```
java -ea Test   # enable assertions
```

20.3 Use Cases of Assertions

- Debugging during development.
- Checking impossible conditions.
- Internal invariants check.

☞ **Note:** Production code में assertions को input validation के लिए use नहीं करना चाहिए।

CH 21 — Best Practices in Exception Handling

- Always use specific exception classes (not just `Exception`).
- Don't leave `catch` blocks empty.
- Use `finally` (or `try-with-resources`) for cleanup.
- Don't overuse checked exceptions (design carefully).
- Don't use exceptions for normal program flow.

CH 22 — Practice Questions (Exception + Assertions)

1. Difference between **throw** and **throws**?
2. Checked vs Unchecked exceptions with examples?
3. Why is `finally` block always executed?
4. Can we have `try` without `catch`? (Yes, with `finally`)
5. What is the difference between Error and Exception?
6. Write a program to handle multiple exceptions in one catch block.
7. What are assertions? How do we enable them in Java?

💡 Part C में हमने cover किया:

- Exception Handling (Hierarchy, Keywords, Examples)
- throw vs throws
- Custom Exceptions
- Assertions (Syntax + Use cases)
- Best Practices + Interview Questions

☞ **Part D (Multithreading & Concurrency)** । इसमें Threads, Synchronization, Deadlock, Concurrency Utilities सब detail में आएगा।

📘 JAVA NOTES — PART D

(Multithreading & Concurrency)

CH 23 — Introduction to Multithreading

23.1 What is Multithreading?

- **Thread** → Lightweight sub-process (independent path of execution)।
- **Multithreading** → एक ही program में multiple threads parallel/concurrently run करते हैं।

☞ Advantage:

- Better CPU utilization
- Faster execution
- Useful in Games, Web Servers, Real-time apps

23.2 Thread Lifecycle (Diagram Explanation)

```
New (Created) → Runnable → Running → Waiting/Timed Waiting → Terminated
```

- **New**: Thread created, not started yet
- **Runnable**: Ready to run, waiting for CPU
- **Running**: Actively executing
- **Waiting/Timed Waiting**: Suspended, waiting for signal/time
- **Terminated**: Finished

CH 24 — Creating Threads

24.1 By Extending Thread class

```
class MyThread extends Thread {
   public void run() {
      System.out.println("Thread running...");
   }
}
public class Test {
   public static void main(String[] args) {
      MyThread t = new MyThread();
      t.start();   // start() internally calls run()
   }
```

```
}
```

24.2 By Implementing Runnable interface

```
class MyTask implements Runnable {
   public void run() {
      System.out.println("Runnable running...");
   }
}
class Test {
   public static void main(String[] args) {
      Thread t = new Thread(new MyTask());
      t.start();
   }
}
```

CH 25 — Thread Methods

- `start()` → Thread start करता है (run() call करता है)।
- `run()` → Thread body।
- `sleep(ms)` → Thread को delay करना।
- `join()` → Wait for thread to finish।
- `yield()` → CPU को छोड़ना ताकि अन्य thread run कर सके।
- `getName(), setName()` → Thread name set/get।
- `setPriority(int)` → Priority (1–10)।
- `interrupt()` → Thread को signal करना।

CH 26 — Synchronization

26.1 Problem: Race Condition

- जब multiple threads एक ही resource access करते हैं → inconsistent result possible।

26.2 Solution: synchronized keyword

Method Level Sync

```
class Counter {
   private int count = 0;
   public synchronized void increment() { count++; }
   public int getCount(){ return count; }
}
```

Block Level Sync

```
synchronized(this) {
   // critical section
}
```

CH 27 — Inter-Thread Communication

- Methods: `wait()`, `notify()`, `notifyAll()`
- Must be called inside synchronized block.

Producer-Consumer Example (Concept):

- Producer thread → data produce करता है।
- Consumer thread → data consume करता है।
- Synchronization + wait/notify use होता है।

CH 28 — Deadlock, Starvation, Livelock

28.1 Deadlock

- जब दो threads resources lock करके एक-दूसरे का wait करें।

```
Thread-1 → lock A → waiting for B
Thread-2 → lock B → waiting for A
```

28.2 Starvation

- Low-priority thread को CPU नहीं मिलता क्योंकि high-priority हमेशा occupy कर लेते हैं।

28.3 Livelock

- Threads continuously react to each other → progress नहीं होता।

☞ Avoid by proper lock ordering, timeouts, avoiding nested locks।

CH 29 — Daemon Threads

- Background threads (e.g., Garbage Collector)।
- JVM इन्हें automatically kill कर देता है जब सारे **user threads** खत्म हो जाते हैं।

```
Thread t = new Thread(...);
t.setDaemon(true);
t.start();
```

CH 30 — Thread Pools (Executor Framework)

- Creating new thread हर बार expensive होता है।
- Solution → **Thread Pool**: Pre-created threads reuse होते हैं।

```
ExecutorService ex = Executors.newFixedThreadPool(5);
for(int i=0;i<10;i++) {
   ex.execute(() -> System.out.println("Task by " +
Thread.currentThread().getName()));
}
ex.shutdown();
```

CH 31 — Concurrency Utilities (java.util.concurrent)

31.1 Lock Framework

- **ReentrantLock** → explicit lock/unlock control।
- **ReadWriteLock** → multiple readers, single writer.

31.2 Atomic Variables

- `AtomicInteger, AtomicLong` etc.
- Thread-safe updates without synchronization.

31.3 Synchronizers

- **CountDownLatch** → wait until certain tasks finish.
- **CyclicBarrier** → multiple threads wait at barrier point.
- **Semaphore** → control no. of threads accessing resource.

31.4 Concurrent Collections

- `ConcurrentHashMap`, `CopyOnWriteArrayList`, `BlockingQueue`.
- Safe for multithreaded environment.

CH 32 — Volatile Keyword

- Declaring a variable **volatile** → ensures visibility of changes across threads.

```
volatile boolean flag = true;
```

☞ Use for simple flags, but **not** for atomic compound operations.

CH 33 — Best Practices in Multithreading

- Use **ExecutorService** instead of manually managing threads.
- Use **Concurrent collections** instead of synchronized collections.
- Avoid unnecessary synchronization (may cause performance drop).
- Always shutdown ExecutorService after use.
- Prefer **immutable objects** in multi-threaded apps.

CH 34 — Practice Questions (Multithreading & Concurrency)

1. Difference between **Thread class** and **Runnable interface**?
2. Explain Thread Life Cycle with diagram.
3. What is the difference between **sleep()** and **wait()**?
4. What is a Daemon thread? Example?
5. What is Deadlock? How to avoid it?
6. Difference between **synchronized method** and **synchronized block**?
7. Explain **volatile** keyword.
8. Difference between **ExecutorService** and traditional threads?
9. What are **CountDownLatch** and **CyclicBarrier**?
10. What is the difference between **ConcurrentHashMap** and **Hashtable**?

❢ Part D में हमने cover किया:

- Multithreading basics + Thread lifecycle
- Thread creation methods
- Synchronization & inter-thread communication
- Deadlock, starvation, livelock
- Daemon threads & Thread pools
- Concurrency utilities (`Locks, Atomic, Executors`)
- Volatile keyword
- Best practices + Interview Qs

☞ **Part E (Collections & Generics)** । इसमें Java Collections Framework का पूरा detail + Generics की depth होगी।

📘 JAVA NOTES — PART E

(Collections & Generics)

CH 35 — Collections Framework Introduction

35.1 What is Collection?

- A **collection** = group of objects (data structures in Java)।
- Java Collections Framework (JCF) = **interfaces + classes + algorithms**।
- Package: `java.util`

👉 Full Forms:

- **JCF** = Java Collections Framework
- **API** = Application Programming Interface

35.2 Advantages of Collections Framework

- Reusable data structures
- Reduces coding effort (ready-made classes)
- Improves performance (optimized algorithms)
- Thread-safe variants available

35.3 Collections Hierarchy (Diagram Explanation)

```
Collection (Interface)
│
├── List
│      ├── ArrayList
│      ├── LinkedList
│      └── Vector → Stack
│
├── Set
│      ├── HashSet
│      ├── LinkedHashSet
│      └── TreeSet
│
└── Queue
       ├── PriorityQueue
       └── Deque → ArrayDeque
```

- **Map (separate hierarchy)**

```
Map
├── HashMap
├── LinkedHashMap
├── TreeMap
└── Hashtable
```

CH 36 — List Interface

- **Ordered** collection, allows **duplicate elements**.
- Index-based access.

36.1 ArrayList

- Dynamic array, fast random access (O(1)).
- Slow in insertion/deletion (middle elements).

```
List<String> list = new ArrayList<>();
list.add("A");
list.add("B");
```

36.2 LinkedList

- Doubly linked list, efficient insert/delete (O(1)).
- Slower random access.

36.3 Vector (Legacy)

- Synchronized (thread-safe).
- Rarely used.

36.4 Stack

- LIFO (Last In First Out).

```
Stack<Integer> st = new Stack<>();
st.push(10);
st.pop();
```

CH 37 — Set Interface

- **Unordered collection**, no duplicates.

37.1 HashSet

- Uses **HashMap internally**.
- Null element allowed (only once).

37.2 LinkedHashSet

- Maintains insertion order.

37.3 TreeSet

- Sorted order (Red-Black Tree).
- Null not allowed.

CH 38 — Map Interface

- Stores data as **Key-Value pairs**.
- Duplicate keys not allowed, values can be duplicate.

38.1 HashMap

- Unordered, allows one null key & multiple null values.

```
Map<Integer,String> map = new HashMap<>();
map.put(1,"A");
map.put(2,"B");
```

38.2 LinkedHashMap

- Maintains insertion order.

38.3 TreeMap

- Sorted map (keys in natural order).
- Null key not allowed.

38.4 Hashtable (Legacy)

- Synchronized, no null key or value allowed.

CH 39 — Queue & Deque

- **Queue (FIFO)** → `LinkedList`, `PriorityQueue`
- **Deque (Double-ended queue)** → `ArrayDeque`

```
Queue<Integer> q = new LinkedList<>();
q.add(10);
q.add(20);
q.poll();  // removes head
```

CH 40 — Utility Classes

- **Collections Class** (algorithms for collections)

```
Collections.sort(list);
Collections.reverse(list);
Collections.shuffle(list);
Collections.max(list);
```

- **Arrays Class** (array operations)

```
Arrays.sort(arr);
Arrays.binarySearch(arr, key);
```

CH 41 — Comparable vs Comparator

Comparable

- Used for **natural ordering**.
- Implemented inside the class itself.

```
class Student implements Comparable<Student> {
   int marks;
   public int compareTo(Student s){ return this.marks - s.marks; }
}
```

Comparator

- Used for **custom ordering**.
- Defined separately.

```
Comparator<Student> byName = (s1,s2) -> s1.name.compareTo(s2.name);
```

CH 42 — Fail-Fast vs Fail-Safe Iterators

- **Fail-Fast (e.g., ArrayList, HashMap)**
  - Throws `ConcurrentModificationException` if structure modified during iteration.
- **Fail-Safe (e.g., CopyOnWriteArrayList, ConcurrentHashMap)**
  - Iterates over a clone, safe against modification.

CH 43 — Generics in Java

43.1 What is Generics?

- Generics = Type-safe data structures.
- Benefit: Compile-time type checking, no casting required.

```
List<String> list = new ArrayList<>();
list.add("Hello");
String s = list.get(0);  // No cast needed
```

43.2 Generic Class

```
class Box<T> {
   private T data;
   public void set(T data){ this.data = data; }
   public T get(){ return data; }
}
Box<Integer> b = new Box<>();
b.set(10);
```

43.3 Bounded Type Parameters

```
class Calc<T extends Number> {
   T num;
}
```

43.4 Wildcards in Generics

- `<?>` → unknown type
- `<? extends T>` → upper bounded (T या T की subclasses)
- `<? super T>` → lower bounded (T या T के superclasses)

CH 44 — Best Practices in Collections & Generics

- Prefer **interfaces (List, Set, Map)** over implementation classes.
- Use **Generics** for type safety.
- For multithreaded apps → use `ConcurrentHashMap`, `CopyOnWriteArrayList`.
- Use `Collections.unmodifiableList()` for read-only collections.
- Avoid `Vector` and `Hashtable` (legacy).

CH 45 — Practice Questions (Collections & Generics)

1. Difference between ArrayList and LinkedList?
2. Difference between HashMap and Hashtable?
3. Why String is good as key in HashMap?
4. Difference between Comparable and Comparator?
5. Explain fail-fast vs fail-safe iterators.
6. What is the use of Generics? Example?
7. What is the difference between `<? extends T>` and `<? super T>`?
8. Why Map is not a part of Collection interface?
9. How HashSet internally works?
10. What are the advantages of ConcurrentHashMap over HashMap?

💡 **Part E में हमने cover किया:**

- Collections Framework hierarchy
- List, Set, Map, Queue implementations
- Utility classes (Collections, Arrays)
- Comparable vs Comparator
- Fail-fast vs Fail-safe
- Generics (Basics, Bounded, Wildcards)
- Best Practices + Interview Qs

☞ **Part F (Modern Java Features — Java 5 से Java 17 तक)।** इसमें हर major Java version की important features detail में होंगी ।

📘 JAVA NOTES — PART F

(Modern Java Features: Java 5 → Java 17)

CH 46 — Java 5 Features (2004)

☞ Nickname: **Tiger Release**

1. **Generics**
   - Type safety for collections & methods

```
List<String> list = new ArrayList<>();
```

2. **Enhanced for loop (for-each)**

```
for(String s : list) { ... }
```

3. **Autoboxing/Unboxing**
   - Automatic conversion between primitive & wrapper

```
Integer x = 5;    // autoboxing
int y = x;        // unboxing
```

4. **Typesafe Enums**

```
enum Day { MON, TUE, WED }
```

5. **Varargs (Variable Arguments)**

```
int sum(int... nums) { ... }
```

6. **Annotations**
   - `@Override`, `@Deprecated`, `@SuppressWarnings`

CH 47 — Java 6 Features (2006)

- Mostly **performance improvements**.
- Scripting support via **javax.script API** (JSR 223).
- JDBC 4.0 auto-loading of drivers.
- `Compiler API` (`javax.tools`) → compile code dynamically.

CH 48 — Java 7 Features (2011)

☞ Nickname: **Dolphin**

1. **Diamond Operator (<>)**

```
List<String> list = new ArrayList<>();
```

2. **try-with-resources**
   - Auto-closeable resources

```
try (BufferedReader br = new BufferedReader(new
FileReader("file.txt"))) {
    System.out.println(br.readLine());
}
```

3. **Catching Multiple Exceptions**

```
try { ... }
catch (IOException | SQLException ex) { ... }
```

4. **String in switch**

```
switch(day) { case "MON": ... }
```

5. **Binary Literals & Underscores in numbers**

```
int bin = 0b1010;
int big = 1_000_000;
```

CH 49 — Java 8 Features (2014)

☞ Most Revolutionary Release

1. **Lambda Expressions**

```
list.forEach(x -> System.out.println(x));
```

2. **Functional Interfaces**
   - `@FunctionalInterface`
   - Example: Runnable, Comparator
3. **Stream API**

```
list.stream().filter(x -> x>10).map(x ->
x*x).forEach(System.out::println);
```

4. **Default & Static methods in Interfaces**

```
interface A {
   default void show() { System.out.println("default"); }
   static void help() { System.out.println("static"); }
}
```

5. **java.time API (Date/Time)**

```
LocalDate d = LocalDate.now();
```

6. **Optional Class**

```
Optional<String> opt = Optional.of("Hello");
```

7. **Nashorn JavaScript Engine**

CH 50 — Java 9 Features (2017)

1. **Modules (Project Jigsaw)**
   - `module-info.java` for modularization.
2. **JShell (REPL)**
   - Interactive execution.
3. **Private methods in Interfaces**
4. **Factory Methods for Collections**

```
List<Integer> nums = List.of(1,2,3);
```

CH 51 — Java 10 Features (2018)

1. **Local Variable Type Inference (`var`)**

```
var msg = "Hello"; // compiler infers String
```

2. Garbage Collector improvements.

CH 52 — Java 11 Features (2018 LTS)

1. **New String Methods**
   - `isBlank()`, `lines()`, `strip()`, `repeat()`
2. **HttpClient API** (standardized)
3. `HttpClient client = HttpClient.newHttpClient();`
4. Run single-file programs without compilation
5. `java Hello.java`

CH 53 — Java 12–14 Features

1. **Switch Expressions (Java 12/14)**

```
int result = switch(day) {
```

```
    case "MON" -> 1;
    case "TUE" -> 2;
    default -> 0;
};
```

2. **Text Blocks (Java 13/14)**

```
String json = """
    {
      "name": "Ram",
      "age": 25
    }
""";
```

3. Pattern Matching for `instanceof` (Java 14)

```
if(obj instanceof String s) {
    System.out.println(s.toLowerCase());
}
```

CH 54 — Java 15 Features (2020)

1. **Sealed Classes (Preview)**

```
sealed class Shape permits Circle, Square { }
final class Circle extends Shape { }
```

2. **Hidden Classes** (used in frameworks).

CH 55 — Java 16 Features (2021)

1. **Records** (immutable data carrier classes)

```
record Person(String name, int age) {}
```

2. **Pattern Matching for instanceof** finalized.

CH 56 — Java 17 Features (2021 LTS)

1. **Sealed Classes (finalized)**
2. **Pattern Matching for Switch (Preview)**

```
switch(obj) {
    case String s -> System.out.println("String " + s);
    case Integer i -> System.out.println("Int " + i);
    default -> System.out.println("Other");
}
```

3. **Strong Encapsulation of JDK internals**
4. Long-Term Support (LTS).

CH 57 — Best Practices with Modern Java

- Use **var** for readability (but not overuse).
- Prefer **Streams & Lambdas** for concise code.
- Use **Optional** to avoid NullPointerException.
- Prefer **Records** for DTOs (Data Transfer Objects).
- Adopt **Modules** for large applications.

CH 58 — Practice Questions (Java 5 → 17 Features)

1. What are Generics? Why introduced in Java 5?
2. What is try-with-resources? Example.
3. Explain Lambda Expressions and Functional Interfaces.
4. Difference between Stream API and Collections API?
5. What is Optional class in Java 8?
6. What is JShell?
7. What is var in Java 10? Difference from JavaScript var?
8. Difference between Records and normal classes?
9. What are Sealed Classes? Why used?
10. What is the difference between Java 8 Streams and Java 17 Pattern Matching?

❗ Part F में हमने cover किया:

- Java 5 → Generics, Enums, Autoboxing
- Java 7 → Diamond, try-with-resources, String in switch
- Java 8 → Lambdas, Streams, Optional, java.time
- Java 9 → Modules, JShell
- Java 10–17 → var, HttpClient, Text Blocks, Records, Sealed Classes, Pattern Matching

---

☞ **Part G (Java APIs & Tools — File I/O, Streams, JDBC, Serialization, Logging)** । इसमें File I/O, Streams API, Date & Time, JDBC, Serialization, Logging सब detail में आएगा।

📘 JAVA NOTES — PART G

(Java APIs & Tools)

CH 59 — File I/O in Java

59.1 Old I/O (java.io package)

- **File class** → represents file/directory.

```
File f = new File("data.txt");
System.out.println(f.exists());
```

- **FileReader / FileWriter** → character streams.

```
FileWriter fw = new FileWriter("data.txt");
fw.write("Hello Java");
fw.close();
```

- **BufferedReader / BufferedWriter** → efficient reading/writing.

```
BufferedReader br = new BufferedReader(new FileReader("data.txt"));
String line = br.readLine();
```

- **PrintWriter** → formatted output.

59.2 New I/O (NIO.2 — java.nio.file, Java 7+)

```
Path path = Paths.get("data.txt");
Files.write(path, "Hello".getBytes());
List<String> lines = Files.readAllLines(path);
```

- Advantages: Better performance, symbolic link support, asynchronous I/O.

CH 60 — Serialization

- Process of converting object → byte stream (save/transfer).
- **Serializable interface** (marker).

```
class Student implements Serializable {
   int id; String name;
}
```

Writing Object

```
ObjectOutputStream out = new ObjectOutputStream(new
FileOutputStream("obj.dat"));
out.writeObject(new Student());
```

Reading Object

```
ObjectInputStream in = new ObjectInputStream(new
FileInputStream("obj.dat"));
Student s = (Student) in.readObject();
```

☞ **transient keyword** → skip variable during serialization.

CH 61 — Streams API (Java 8)

- **Stream = sequence of elements supporting functional-style operations.**

61.1 Operations

- Intermediate → `map`, `filter`, `sorted`
- Terminal → `collect`, `forEach`, `reduce`

61.2 Example

```
List<Integer> nums = Arrays.asList(2,4,6,7,8);
List<Integer> evenSquares =
   nums.stream()
       .filter(x -> x%2==0)
       .map(x -> x*x)
       .collect(Collectors.toList());
```

CH 62 — Date & Time API (java.time, Java 8)

- Old `Date`, `Calendar` confusing → replaced by **java.time**.

62.1 LocalDate, LocalTime, LocalDateTime

```
LocalDate d = LocalDate.now();
LocalDate birth = LocalDate.of(1995, 5, 10);
```

62.2 Period & Duration

```
Period p = Period.between(birth, d);
Duration dur = Duration.ofHours(5);
```

62.3 DateTimeFormatter

```
DateTimeFormatter f = DateTimeFormatter.ofPattern("dd-MM-yyyy");
System.out.println(d.format(f));
```

CH 63 — JDBC (Java Database Connectivity)

☞ Full Form: JDBC = **Java Database Connectivity**

63.1 Steps to Connect with DB

1. **Load Driver**

```
Class.forName("com.mysql.cj.jdbc.Driver");
```

2. **Create Connection**

```
Connection con = DriverManager.getConnection(
   "jdbc:mysql://localhost:3306/test","root","password");
```

3. **Create Statement**

```
PreparedStatement ps = con.prepareStatement("insert into emp values(?,?)");
ps.setInt(1, 101);
ps.setString(2, "Ram");
```

4. **Execute Query**

```
ps.executeUpdate(); // insert/update/delete
ResultSet rs = ps.executeQuery(); // select
```

5. **Close Connection**

```
con.close();
```

63.2 Types of JDBC Drivers

1. Type-1 → JDBC-ODBC Bridge (obsolete)
2. Type-2 → Native API
3. Type-3 → Network Protocol
4. Type-4 → Thin Driver (Pure Java, most used)

CH 64 — Logging API

- **java.util.logging** (default).
- Popular external: **Log4j, SLF4J, Logback**.

Example

```
import java.util.logging.*;
class Test {
   private static Logger logger = Logger.getLogger("MyLog");
   public static void main(String[] args) {
      logger.info("Info Message");
      logger.warning("Warning Message");
   }
}
```

CH 65 — Java Annotations (Advanced Use)

- **Built-in:** `@Override`, `@Deprecated`, `@SuppressWarnings`
- **Meta-annotations:**
  - `@Retention` → how long annotation available
  - `@Target` → where annotation can be applied

Custom Annotation Example

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@interface MyAnnotation { String value(); }

class Demo {
   @MyAnnotation("test")
   public void show() { }
}
```

CH 66 — Best Practices (APIs & Tools)

- Always close resources (try-with-resources).
- Use **PreparedStatement** instead of Statement (SQL Injection prevention).
- Prefer `java.time` API instead of `Date/Calendar`.
- Use logging framework (Log4j/SLF4J) instead of `System.out.println`.
- Serialize only when required (avoid security issues).

CH 67 — Practice Questions (APIs & Tools)

1. Difference between FileReader and BufferedReader?
2. What is Serialization? What is `transient` keyword?

3. What is the difference between Stream API and I/O Streams?
4. Explain `LocalDate` vs `Date`.
5. Explain steps to connect Java with MySQL using JDBC.
6. What are the different types of JDBC drivers?
7. What is the difference between Statement and PreparedStatement?
8. What is the use of logging API?
9. Explain custom annotations with example.
10. Why try-with-resources is better than finally?

💡 **Part G में हमने cover किया:**

- File I/O (Old & NIO.2)
- Serialization & transient
- Streams API (Java 8)
- Date & Time API (java.time)
- JDBC (Steps, Drivers)
- Logging API (java.util.logging + external)
- Annotations (built-in, custom)
- Best Practices + Interview Qs

☞

Part H (Best Practices, Pitfalls & Advanced Interview Topics) । इसमें Java coding standards, common mistakes, security issues और interview में बार-बार पूछे जाने वाले tricky concepts cover होंगे।

📘 JAVA NOTES — PART H

(Best Practices, Pitfalls & Advanced Interview Topics)

CH 68 — Java Coding Standards (Best Practices)

68.1 Naming Conventions

- Classes/Interfaces → **PascalCase** (`EmployeeDetails`)
- Methods/Variables → **camelCase** (`calculateSalary`)
- Constants → **UPPER_CASE** (`MAX_VALUE`)
- Packages → **lowercase** (`com.company.project`)

68.2 Code Structure

- One class = one file (public class = file name).
- Indentation, comments, Javadoc for public APIs.

68.3 Design Guidelines

- Prefer **interfaces** over concrete classes.
- Use **composition** over inheritance.
- Keep methods **short & focused**.
- Apply **SOLID principles** (OOP design).

CH 69 — Performance Best Practices

- Use **StringBuilder** instead of string concatenation in loops.
- Prefer **ArrayList** for search, **LinkedList** for frequent insert/delete.
- Use **HashMap** for lookup, avoid Hashtable (legacy).
- Use **ExecutorService** instead of creating new threads.
- Release unused references → help Garbage Collector.

CH 70 — Memory Management Pitfalls

- **Memory Leak** → जब unused objects referenced रह जाते हैं।
- Avoid static references for large objects.
- Prefer local variables over static for temporary use.
- Always close streams, sockets, DB connections.

CH 71 — Common Java Pitfalls

71.1 String Comparison

```
String s1 = new String("Hello");
String s2 = new String("Hello");
System.out.println(s1 == s2);       // false
System.out.println(s1.equals(s2));  // true ✅
```

71.2 Floating Point Precision

```
System.out.println(0.1 + 0.2); // 0.30000000000000004
```

👉 Use **BigDecimal** for financial calculations.

71.3 NullPointerException (NPE)

- Common cause: dereferencing null.
- Avoid with `Objects.requireNonNull()`, `Optional<T>`.

71.4 ConcurrentModificationException

- Happens with fail-fast collections during iteration.
- Solution → Use `Iterator.remove()` or `CopyOnWriteArrayList`.

CH 72 — Security Best Practices

- Always use **PreparedStatement** to prevent SQL Injection.
- Validate inputs (never trust user input).
- Avoid **Object Serialization** for sensitive data (can be tampered).
- Use **encryption** for passwords, not plain text.
- Use **immutable classes** for critical objects (thread safety).

CH 73 — Advanced Keywords

73.1 volatile

- Variable changes are visible across threads.

73.2 transient

- Skip during serialization.

73.3 strictfp

- Floating point calculations strict, portable across platforms.

73.4 native

- For methods implemented in non-Java code (C/C++).

CH 74 — Important "Why" Questions

1. **Why String is immutable in Java?**
   - Security (used in ClassLoader, DB URL, etc.)
   - Caching in String pool
   - Thread-safety
2. **Why multiple inheritance not supported in Java (classes)?**
   - Diamond problem (ambiguity).
   - Solved using **interfaces**.
3. **Why Java is platform independent?**
   - Compiles to **bytecode** → runs on JVM of any OS.
4. **Why main() is static?**
   - JVM can call it without creating object.

CH 75 — Important "Difference Between"

- **throw vs throws**
- **final vs finally vs finalize**
- **== vs equals()**
- **HashMap vs Hashtable**
- **ArrayList vs Vector**
- **Comparable vs Comparator**
- **Checked vs Unchecked exceptions**
- **String vs StringBuffer vs StringBuilder**

CH 76 — Advanced Interview Topics

76.1 Fail-fast vs Fail-safe Iterators

- Fail-fast → throw `ConcurrentModificationException`.
- Fail-safe → work on clone, safe.

76.2 WeakHashMap

- Keys are weak references → eligible for GC if no strong reference exists.

76.3 ClassLoader in Java

- Loads classes at runtime.
- Types: Bootstrap, Extension, Application ClassLoader.

76.4 Reflection API

- Inspect classes, methods, fields at runtime.

```
Class c = Class.forName("java.lang.String");
Method[] methods = c.getDeclaredMethods();
```

76.5 JIT Compiler

- Part of JVM.
- Converts bytecode → native code for faster execution.

CH 77 — Practice Questions (Best Practices + Pitfalls)

1. Why String is immutable in Java?
2. Difference between final, finally, and finalize()?
3. What are memory leaks in Java? How to avoid them?
4. What is difference between fail-fast and fail-safe iterators?
5. What is the use of volatile keyword?
6. Why use PreparedStatement instead of Statement?
7. What is the role of ClassLoader in Java?
8. What are weak references in Java?
9. What is Reflection API? Why is it dangerous?
10. What is the purpose of JIT compiler?

💡 **Part H में हमने cover किया:**

- Coding standards + Design practices
- Performance tuning tips
- Memory management pitfalls
- Common mistakes (Strings, NPE, floating-point, concurrency)
- Security practices (SQL injection prevention, immutability)
- Advanced keywords (volatile, transient, strictfp, native)
- Why Questions + Key Differences + Advanced Topics

☞

Part I (Practice Material — Interview Questions + Coding Exercises + Mini Projects) । इससे आपके Notes पूरी तरह self-contained हो जाएंगे और exam + interview दोनों के लिए काम आएँगे।

📘 JAVA NOTES — PART I

(Practice Material: Interview Qs, Coding Exercises, Mini Projects)

CH 78 — Frequently Asked Interview Questions

78.1 Core Java (Basics)

1. What are the main features of Java?
2. Explain JVM, JRE, and JDK.
3. What is the difference between bytecode and machine code?
4. Why is Java platform independent?
5. Why Java is not purely object-oriented?
6. Explain OOPs concepts with real-world examples.

78.2 Strings & OOPs

1. Difference between String, StringBuffer, and StringBuilder.
2. Why is String immutable?
3. Difference between method overloading and overriding.
4. Can constructor be overridden?
5. Explain access modifiers in Java.

78.3 Exceptions & Multithreading

1. Difference between checked and unchecked exceptions.
2. Difference between throw and throws.
3. Can we have try without catch?
4. What is the difference between wait() and sleep()?
5. What is the difference between process and thread?

78.4 Collections & Generics

1. Difference between ArrayList and LinkedList.
2. Difference between HashMap and Hashtable.
3. What is ConcurrentHashMap?
4. Difference between Comparable and Comparator.
5. What are generics? Advantages?

78.5 Java Advanced & New Features

1. What is Lambda Expression?
2. What is Stream API?
3. What is Optional?
4. What are Records in Java 16?
5. What are Sealed classes in Java 17?

CH 79 — Short Coding Exercises

79.1 Prime Number Check

```
int n=29, i, flag=0;
for(i=2;i<=n/2;i++){
   if(n%i==0){ flag=1; break; }
}
System.out.println(flag==0 ? "Prime" : "Not Prime");
```

79.2 Fibonacci Series

```
int a=0,b=1,c;
for(int i=1;i<=10;i++){
   System.out.print(a+" ");
   c=a+b; a=b; b=c;
}
```

79.3 Palindrome String

```
String s="madam";
String rev=new StringBuilder(s).reverse().toString();
System.out.println(s.equals(rev) ? "Palindrome" : "Not Palindrome");
```

79.4 Factorial using Recursion

```
int fact(int n){ return (n==0)?1:n*fact(n-1); }
```

79.5 File Reading Example

```
try(BufferedReader br=new BufferedReader(new FileReader("data.txt"))){
   br.lines().forEach(System.out::println);
}
```

CH 80 — Advanced Coding Challenges

1. **Find duplicate characters in a string**
2. **Reverse words in a sentence**
3. **Check Anagram Strings**
4. **Find missing number in an array**
5. **Producer-Consumer problem using wait/notify**
6. **Deadlock Simulation Program**
7. **Employee sorting by salary using Comparator**
8. **Multithreaded counter using AtomicInteger**
9. **Student database using JDBC**
10. **JSON Parsing using Jackson/Gson (bonus advanced topic)**

CH 81 — Mini Projects (Practice-Oriented)

81.1 Banking System (OOP + Exception Handling + Collections)

- Features: Create Account, Deposit, Withdraw, Balance Check.
- Use: `HashMap<Integer,Account>` for storing accounts.
- Add: Custom Exception `InsufficientBalanceException`.

81.2 Library Management System (Collections)

- Features: Add Books, Issue Book, Return Book, Search Book.
- Use: `ArrayList<Book>` & `HashMap<Integer,Member>`.

81.3 Student Record Manager (JDBC + Collections)

- Features: Insert, Update, Delete, Search student records in DB.
- Use PreparedStatement for DB operations.

81.4 Multithreaded Chat Simulator (Threads + Sockets)

- Two threads for send & receive messages.
- Useful for networking practice.

81.5 Log Analyzer (File I/O + Streams API)

- Read server log file.
- Count error/warning/info logs.
- Generate summary report.

CH 82 — Mock Interview Questions

1. If Java is platform independent, why JVM is platform dependent?
2. How HashMap works internally (hashing + buckets)?
3. What is Garbage Collection? Can you call GC explicitly?
4. Difference between final, finally, finalize().
5. What is ThreadLocal in Java?
6. What is difference between Callable and Runnable?
7. How does ConcurrentHashMap prevent thread interference?
8. Difference between parallelStream() and stream()?
9. Can we override static methods? Why/Why not?
10. Can a constructor be synchronized?

CH 83 — Revision Strategy

- **Step 1:** Read **Part A–H notes** sequentially (Core → Advanced).
- **Step 2:** Solve **Part I coding exercises** daily (at least 2).
- **Step 3:** Pick 5–10 **interview questions** per day & revise.
- **Step 4:** Implement at least **1 mini project** fully.
- **Step 5:** Keep practicing with **LeetCode / HackerRank** for algorithmic confidence.

💡 **Part I में हमने cover किया:**

- 100+ Frequently asked interview questions
- Short coding exercises (prime, palindrome, factorial, file I/O)
- Advanced coding challenges (threads, collections, JDBC)
- 5 Mini projects for real-world practice
- Mock interview Qs + Revision plan

☞ ये **Parts A से लेकर I तक Java Notes set** है, जिसमें theory + examples + best practices + interview prep + projects सब शामिल है।

- **Part A** → Core Java Foundation
- **Part B** → Advanced OOP & Language Features
- **Part C** → Exception Handling + Assertions
- **Part D** → Multithreading & Concurrency
- **Part E** → Collections & Generics
- **Part F** → Modern Java Features (Java 5–17)
- **Part G** → Java APIs & Tools (I/O, JDBC, Logging, etc.)
- **Part H** → Best Practices & Pitfalls
- **Part I** → Practice Material (Interview Qs + Coding Exercises + Mini Projects)