

Great topic! Let's explore Composition and Aggregation in Java, which represent "Has-A" relationships between classes.

◆ 1. "Has-A" Relationship

When one class contains a reference to another class, it is called a "Has-A" relationship. For example, a Car has an Engine.

This is implemented via object references, and it comes in two forms: composition and aggregation.

◆ 2. Composition (Strong Ownership)

- Represents a strong relationship.
- The contained object cannot exist independently of the container.
- If the container is destroyed, so is the contained object.

✅ Example: A House has Rooms. If the House is destroyed, Rooms don't exist independently.

```
class Room {
    private String name;

    Room(String name) {
        this.name = name;
    }
}

class House {
    private Room room;

    House() {
        room = new Room("Living Room"); // Owned and created inside
    }
}
```

👉 In composition, the containing class creates and manages the lifecycle of the contained object.

◆ 3. Aggregation (Shared Relationship)

- Represents a weaker relationship.
- The contained object can exist independently.
- The container just holds a reference.

✅ Example: A Library has Books, but a Book can exist outside any Library.

```
class Book {
    private String title;
```

```

    Book(String title) {
        this.title = title;
    }
}

class Library {
    private List<Book> books;

    Library(List<Book> books) {
        this.books = books; // Book objects passed from outside
    }
}

```

👉 In aggregation, the contained object is not owned by the container and can be shared.

🗨️ Summary Table

Feature	Composition	Aggregation
Relationship Type	Strong “has-a”	Weak “has-a”
Lifecycle	Contained object depends on owner	Contained object lives independently
Example	House → Room	Library → Book
Reusability	No (tightly bound)	Yes (can be shared)

Here's a concise overview of Encapsulation in Java and best practices related to it:

✅ Encapsulation Best Practices

Encapsulation is one of the four pillars of OOP. It means "wrapping data (variables) and code (methods) together as a single unit," and controlling access to that data.

🔒 1. Data Hiding via Private Fields

- Declare fields as `private` so they can't be accessed directly from outside the class.
- Protects the internal state of the object.

```

public class Person {
    private String name; // Hidden from outside
    private int age;
}

```

🔒 2. Providing Controlled Access via Getters and Setters

- Use `public` methods to provide controlled access.
- You can add validation, logging, or other logic in setters.

```
public class Person {
    private String name;
    private int age;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

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

3. Immutability and final Keyword

Immutability means the object's state cannot change after creation.

To make a class immutable:

- Make class `final` (so it can't be subclassed).
- Make fields `private` and `final`.
- Do not provide setters.
- Initialize fields in the constructor only.

```
public final class Employee {
    private final String id;
    private final String name;

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

    public String getId() {
        return id;
    }

    public String getName() {
        return name;
    }
}
```

 This protects the object from modification after it's created.

✓ Summary

Practice	Benefit
private fields	Data hiding
public getters/setters	Controlled access
final keyword & no setters	Immutability
Encapsulation	Improves maintainability and security

Great question. Let's break it down with a simple explanation and example.

✦ What is hashCode() used for?

The hashCode() method is used when Java objects are stored in hash-based collections like:

- HashMap
- HashSet
- Hashtable

These collections use the hashCode() to quickly find where to store or locate an object.

Think of hashCode() as a way to turn an object into a number that tells Java: "Put this object in bucket #N."

⚠ Important Rule:

- If two objects are equal according to equals(), they must return the same hashCode().
- But if two objects have the same hashCode(), they are not necessarily equal (just in the same bucket).

💡 Example Without hashCode() and equals():

Imagine this class:

```
class Student { String name; Student(String name) { this.name = name; } }
```

Now try this:

```
Set set = new HashSet<>(); set.add(new Student("Alice"));
System.out.println(set.contains(new Student("Alice"))); // false!
```

Why? Because each new Student("Alice") has a different hashCode and equals() behavior (from Object).

✓ Fix: Override equals() and hashCode():

```
class Student { String name; Student(String name) { this.name = name; } }
```

```
@Override
public boolean equals(Object o) {
```

```

    if (this == o) return true;
    if (!(o instanceof Student)) return false;
    Student s = (Student) o;
    return name.equals(s.name);
}

@Override
public int hashCode() {
    return name.hashCode();
}
}

```

```

}

```

Now:

```

Set set = new HashSet<>(); set.add(new Student("Alice"));
System.out.println(set.contains(new Student("Alice"))); // true!

```

Great! Let's go through the first three SOLID principles in Object-Oriented Programming (OOP) design using Java-friendly explanations and examples.

◆ SOLID Overview

SOLID is an acronym for five key principles that help you design better, maintainable, and flexible object-oriented software:

1. S → Single Responsibility Principle (SRP)
2. O → Open/Closed Principle (OCP)
3. L → Liskov Substitution Principle (LSP)
4. I → Interface Segregation Principle
5. D → Dependency Inversion Principle

We'll cover the first three here:

1. Single Responsibility Principle (SRP)

"A class should have only one reason to change."

 That means: One class = One job.

 Example:

Bad design:

```

class Report {
    public void generateReport() { /* ... */ }
    public void saveToFile() { /* ... */ } // violates SRP
}

```

Better design:

```

class Report {
    public void generateReport() { /* ... */ }
}

```

```
}

class ReportSaver {
    public void saveToFile(Report report) { /* ... */ }
}
```

Now, each class has one responsibility: generating or saving.

2. Open/Closed Principle (OCP)

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

 That means: You should be able to add new behavior without changing existing code.

 Example:

Bad design:

```
class AreaCalculator {
    public double calculateArea(Object shape) {
        if (shape instanceof Circle) { /* ... */ }
        else if (shape instanceof Rectangle) { /* ... */ }
        // What if we add Triangle? Must modify this class. Bad.
    }
}
```

Better with OCP using inheritance:

```
abstract class Shape {
    abstract double area();
}

class Circle extends Shape {
    double radius;
    public double area() { return Math.PI * radius * radius; }
}

class Rectangle extends Shape {
    double width, height;
    public double area() { return width * height; }
}

class AreaCalculator {
    public double calculateArea(Shape shape) {
        return shape.area(); // no need to modify this when new shapes are
        added
    }
}
```

3. Liskov Substitution Principle (LSP)

"Subtypes must be substitutable for their base types."

📦 That means: If class B is a subclass of class A, we should be able to replace A with B without breaking anything.

📄 Example:

```
class Bird {
    public void fly() {
        System.out.println("Bird is flying");
    }
}

class Ostrich extends Bird {
    @Override
    public void fly() {
        throw new UnsupportedOperationException("Ostriches can't fly!");
    }
}
```

💡 This violates LSP because Ostrich is not really a substitute for Bird if we expect all Birds to fly.

✅ Better design:

```
interface Flyable {
    void fly();
}

class Bird { }

class Sparrow extends Bird implements Flyable {
    public void fly() {
        System.out.println("Sparrow flies");
    }
}

class Ostrich extends Bird {
    // no fly() method
}
```

Now, only birds that can fly implement the Flyable interface.

Absolutely! Let's complete the SOLID principles with the final two:

◆ 4. Interface Segregation Principle (ISP)

"Clients should not be forced to depend on interfaces they do not use."

📦 In other words: Create smaller, more specific interfaces rather than large, general-purpose ones.

❌ Bad Design:

```
interface Worker {
    void work();
    void eat();
}

class Robot implements Worker {
    public void work() { System.out.println("Robot working"); }
    public void eat() {
        throw new UnsupportedOperationException("Robots don't eat");
    }
}
```

This violates ISP — Robot is forced to implement a method it doesn't need.

✓ Good Design:

Split the interface:

```
interface Workable {
    void work();
}

interface Eatable {
    void eat();
}

class Human implements Workable, Eatable {
    public void work() { System.out.println("Human working"); }
    public void eat() { System.out.println("Human eating"); }
}

class Robot implements Workable {
    public void work() { System.out.println("Robot working"); }
}
```

Now each class only implements the interfaces relevant to it. ✓

◆ 5. Dependency Inversion Principle (DIP)

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

📦 Meaning: Code should depend on interfaces or abstract classes rather than concrete classes.

✗ Bad Design:

```
class LightBulb {
    void turnOn() { System.out.println("LightBulb on"); }
    void turnOff() { System.out.println("LightBulb off"); }
}
```



```
class Switch {
    private LightBulb bulb;

    public Switch(LightBulb bulb) {
        this.bulb = bulb;
    }

    void operate(String command) {
        if (command.equals("ON")) bulb.turnOn();
        else bulb.turnOff();
    }
}
```

Switch is tightly coupled to LightBulb.

✓ Good Design:

Use an interface:

```
interface Switchable {
    void turnOn();
    void turnOff();
}

class LightBulb implements Switchable {
    public void turnOn() { System.out.println("LightBulb on"); }
    public void turnOff() { System.out.println("LightBulb off"); }
}

class Fan implements Switchable {
    public void turnOn() { System.out.println("Fan on"); }
    public void turnOff() { System.out.println("Fan off"); }
}

class Switch {
    private Switchable device;

    public Switch(Switchable device) {
        this.device = device;
    }

    void operate(String command) {
        if (command.equals("ON")) device.turnOn();
        else device.turnOff();
    }
}
```

Now, Switch can work with any Switchable device (fan, bulb, etc). This promotes flexibility and testability.

Great question!

When we say a **static nested class** in Java:

"Behaves like a top-level class but scoped within the outer class,"

here's what it really means, broken down simply:

✅ "Behaves like a top-level class"

This means:

- You can create an object of it **without creating the outer class's object**.
 - It can have its own **methods, variables, constructors**, just like any other class.
 - It behaves independently (like a regular class).
-

✅ "Scoped within the outer class"

This means:

- It is **logically grouped** inside the outer class for better organization.
 - Its **full name** includes the outer class name: e.g.,
`OuterClass.StaticNestedClass`
 - It **cannot exist on its own** — you must access it through the outer class.
-

💡 Think of it like:

Imagine a **class inside a folder** (outer class). It works like any regular class but is **organized inside** that folder, so you access it as:

```
OuterClass.StaticNestedClass
```

🔧 Example Recap:

```
OuterClass.StaticNestedClass obj = new OuterClass.StaticNestedClass();
```

You didn't need:

```
OuterClass outer = new OuterClass(); // Not required
```

You're absolutely right! Let's break down **Anonymous Inner Classes** with a **simple explanation and example**.

✅ What is an Anonymous Inner Class?

- It is a **class without a name**, defined and instantiated in **one line**.
 - Mostly used to **override a method** or **implement an interface on the spot**.
 - Very common in:
 - **GUI event handling**
 - **Threads (Runnable)**
 - **Collections (Comparator)**
-

General Syntax:

```
InterfaceOrClass obj = new InterfaceOrClass() {  
    // override method(s)  
};
```

✅ Example 1: Using Anonymous Class with Runnable

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

✅ Example 2: Anonymous Class with an Abstract Class

```
abstract class Animal {  
    abstract void sound();  
}  
  
public class Main {  
    public static void main(String[] args) {  
        Animal dog = new Animal() {  
            void sound() {  
                System.out.println("Bark!");  
            }  
        };  
  
        dog.sound(); // Output: Bark!  
    }  
}
```

Key Points:

Feature	Description
No class name	Defined inline, not reused
Used for 1-time use cases	Often in event listeners or quick method overrides
Can extend abstract class or interface	Yes
Can access final or effectively final variables	Yes (from enclosing method)

LAMBDA

<https://youtu.be/tj5sLSFjVj4?si=dLboLtGvI-JsBa66>

TOPIC: 10 Input Output Stream

✓ 1. Definition

I/O streams in Java are used to perform **input and output operations**. They help in **reading data from** or **writing data to** various sources like files, memory, network, etc.

- Java supports two main types of data:
 - **8-bit** (byte-level): for binary data like images, audio, etc.
 - **16-bit** (char-level): for character/text data (Unicode characters).
-

✓ 2. Types of Streams

◆ Byte Streams

- Used for **binary data** (e.g., images, PDFs, audio).
- Handle data as **8-bit bytes**.
- Main abstract classes:
 - `InputStream` – for reading bytes
 - `OutputStream` – for writing bytes

✓ Example:

```
FileInputStream fin = new FileInputStream("file.pdf");
FileOutputStream fout = new FileOutputStream("output.pdf");
```

◆ Character Streams

- Used for **text data** (i.e., characters).
- Handle data as **16-bit Unicode characters**.
- Main abstract classes:
 - `Reader` – for reading characters
 - `Writer` – for writing characters

✓ Example:

```
FileReader reader = new FileReader("file.txt");
FileWriter writer = new FileWriter("output.txt");
```

✓ 3. Important Classes

Stream Type	For Reading	For Writing
Byte Streams	<code>FileInputStream</code>	<code>FileOutputStream</code>
Character Streams	<code>FileReader</code>	<code>FileWriter</code>

These classes are commonly used for **file-based I/O**.

✓ 4. Common Methods

Method	Purpose
<code>read()</code>	Reads data (1 byte or 1 char at a time)
<code>write()</code>	Writes data (1 byte or 1 char at a time)
<code>close()</code>	Closes the stream and releases resources
<code>flush()</code>	Forces any buffered output to be written immediately

✓ 5. Try-with-Resources

- A special Java feature (since Java 7) to **automatically close** streams after use.
- Used with classes that implement `AutoCloseable` (e.g., all I/O streams).

✓ Example:

```
try (FileReader reader = new FileReader("file.txt")) {  
    int data;  
    while ((data = reader.read()) != -1) {  
        System.out.print((char) data);  
    }  
} catch (IOException e) {  
    System.out.println("Error: " + e.getMessage());  
}  
// No need to call reader.close(); it's done automatically!
```

📌 Summary

Concept	Use
Byte vs Char	Binary vs Text Data
Streams	<code>InputStream/OutputStream</code> and <code>Reader/Writer</code>
Common Classes	<code>FileInputStream</code> , <code>FileOutputStream</code> , <code>FileReader</code> , etc.

Concept	Use
Common Methods	<code>read()</code> , <code>write()</code> , <code>close()</code> , <code>flush()</code>
Try-with-resources	Simplifies resource cleanup

Here's a clear explanation of **Buffered Streams** in Java, broken down by your points:

✓ 1. Why Buffered Streams?

Buffered streams are used to **improve performance** during I/O operations.

◆ Problem:

- Without buffering, each `read()` or `write()` call accesses the physical disk/device.
- This is **slow**, especially with large files or many small reads/writes.

◆ Solution:

- Buffered streams use an **internal buffer** (e.g., 8192 bytes by default).
 - They read/write **in chunks** rather than byte-by-byte or char-by-char.
 - This reduces the number of physical I/O operations = **faster program**.
-

✓ 2. Important Classes

Purpose	Byte Stream	Character Stream
Reading	<code>BufferedInputStream</code>	<code>BufferedReader</code>
Writing	<code>BufferedOutputStream</code>	<code>BufferedWriter</code>

✓ 3. Key Methods

Class	Key Method	Description
<code>BufferedReader</code>	<code>readLine()</code>	Reads a line of text as a String

`readLine()` is not available in `FileReader`, only in `BufferedReader`.

✓ 4. How It Works

Buffered streams use an **internal byte/char array** (default size: **8192 bytes**):

- When reading:

- It reads a full block of data into memory from disk **at once**.
- Your code then reads from that buffer, not the disk repeatedly.
- When writing:
 - Your data is stored temporarily in a buffer.
 - It only writes to disk when:
 - The buffer is full
 - You manually call `flush()`
 - Or `close()` is called

✓ Example: Using `BufferedReader` and `BufferedWriter`

```
import java.io.*;

public class BufferedExample {
    public static void main(String[] args) {
        // Writing to a file using BufferedWriter
        try (BufferedWriter writer = new BufferedWriter(new
FileWriter("output.txt"))) {
            writer.write("Hello, world!");
            writer.newLine();
            writer.write("Buffered writing is efficient.");
        } catch (IOException e) {
            System.out.println("Write Error: " + e.getMessage());
        }

        // Reading from the file using BufferedReader
        try (BufferedReader reader = new BufferedReader(new
FileReader("output.txt"))) {
            String line;
            while ((line = reader.readLine()) != null) {
                System.out.println("Read: " + line);
            }
        } catch (IOException e) {
            System.out.println("Read Error: " + e.getMessage());
        }
    }
}
```

Summary

Concept	Description
Buffered Streams	Improve performance by minimizing disk access
Key Classes	<code>BufferedReader</code> , <code>BufferedWriter</code> , <code>BufferedInputStream</code> , <code>BufferedOutputStream</code>

Concept	Description
Key Method	<code>readLine()</code> – reads a whole line of text
Internal Buffer	Typically 8192 bytes – reads/writes data in chunks

Here's a detailed explanation of **Object Streams** in Java, covering all the points you mentioned:

✓ 1. Purpose of Object Streams

Object Streams are used to **read and write entire Java objects** to files, memory, or network.

This is called **serialization** – converting an object into a byte stream so it can be:

- Saved to a file
 - Sent over a network
 - Stored for later use
-

✓ 2. Key Classes

Class	Purpose
<code>ObjectOutputStream</code>	Writes (serializes) objects to a stream
<code>ObjectInputStream</code>	Reads (deserializes) objects from a stream

✓ Basic Example:

```
ObjectOutputStream out = new ObjectOutputStream(new
FileOutputStream("data.ser"));
out.writeObject(myObject);
```

✓ 3. Serializable Interface

For an object to be written using `ObjectOutputStream`, its class must **implement** `java.io.Serializable`.

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

- It's a **marker interface** (no methods to implement)
 - Signals the JVM that the class can be serialized
-

✓ 4. transient Keyword

Fields marked as `transient` **will not be serialized**.

Use it for:

- Sensitive data (e.g., passwords)
- Temporary data (e.g., cached values)

```
class User implements Serializable {  
    String name;  
    transient String password;    // Not saved to file  
}
```

✓ 5. Versioning – serialVersionUID

Used to verify the class version during deserialization.

```
private static final long serialVersionUID = 1L;
```

Why important?

- Prevents `InvalidClassException` if the class structure changes after serialization.

If not manually defined, JVM generates one. But if the class changes later, the generated ID will mismatch during deserialization.

✓ 6. Deep Copy Using Object Streams

Object streams can be used to create a **deep copy** of an object (if all fields are serializable).

```
public static <T extends Serializable> T deepCopy(T object) {  
    try {  
        ByteArrayOutputStream bos = new ByteArrayOutputStream();  
        ObjectOutputStream out = new ObjectOutputStream(bos);  
        out.writeObject(object);  
  
        ByteArrayInputStream bis = new  
        ByteArrayInputStream(bos.toByteArray());  
        ObjectInputStream in = new ObjectInputStream(bis);  
  
        return (T) in.readObject();  
    } catch (Exception e) {  
        throw new RuntimeException(e);  
    }  
}
```

✓ Full Example: Serialization and Deserialization

```

import java.io.*;

class Student implements Serializable {
    private static final long serialVersionUID = 1L;
    int id;
    String name;
    transient String password;

    public Student(int id, String name, String password) {
        this.id = id;
        this.name = name;
        this.password = password;
    }
}

public class ObjectOutputStreamExample {
    public static void main(String[] args) {
        Student s1 = new Student(101, "Alice", "secret123");

        // Serialize
        try (ObjectOutputStream oos = new ObjectOutputStream(new
FileOutputStream("student.ser"))) {
            oos.writeObject(s1);
        } catch (IOException e) {
            e.printStackTrace();
        }

        // Deserialize
        try (ObjectInputStream ois = new ObjectInputStream(new
FileInputStream("student.ser"))) {
            Student s2 = (Student) ois.readObject();
            System.out.println("ID: " + s2.id + ", Name: " + s2.name + ",
Password: " + s2.password);
            // Password will be null because it was marked transient
        } catch (IOException | ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}

```

Summary Table



Feature	Explanation
Object Streams	Serialize/deserialize full objects
Key Classes	ObjectInputStream, ObjectOutputStream
Serializable	Interface a class must implement to be serialized
transient	Fields marked won't be saved in serialization
serialVersionUID	Prevents <code>InvalidClassException</code> when class versions differ
Deep Copy	Clone objects by serializing and deserializing

Topic 12 Concurrency





1. Concurrency vs. Parallelism

Concept	Description
Concurrency	Multiple tasks make progress over time, interleaved. (Single CPU can switch between tasks)
Parallelism	Multiple tasks run at the same time on multiple processors/cores.

2. Benefits of Concurrency

-  **Better resource utilization:** CPU and I/O devices can work simultaneously.
-  **Improved application responsiveness:** UI remains responsive during background tasks (e.g., downloads).

3. Challenges of Concurrency

-  **Race Conditions:** Two threads access and modify shared data at the same time.
-  **Deadlocks:** Two threads wait for each other's resources forever.
-  **Starvation:** A thread never gets CPU time because other threads hog resources.
-  **Livelock:** Threads keep changing state in response to each other, but make no progress.

4. Thread Lifecycle in Java

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

- **New:** Thread object created but not started.
- **Runnable:** `start()` is called, waiting for CPU.
- **Running:** Thread is actually executing.
- **Blocked/Waiting:** Thread waiting for lock or signal.
- **Terminated:** Thread has finished execution.

5. Context Switching and Its Cost

- **Context Switching:** The CPU switches between threads by saving and restoring states.
- **Cost:** Involves memory operations and CPU time, which can slow down performance if overused.

6. Critical Section and Synchronization

- **Critical Section:** A block of code that accesses shared resources.
- **Synchronization:** Java uses `synchronized` keyword or locks (`ReentrantLock`) to ensure only one thread executes the critical section at a time.

```
synchronized void increment() {  
    counter++;  
}
```

7. Volatile Keyword

```
volatile boolean running = true;
```

- Guarantees **visibility** of variable changes across threads.
- Does **not** ensure atomicity.
- Useful for flags and read-heavy variables.

8. Thread Safety

- A class is **thread-safe** if it behaves correctly when accessed by multiple threads concurrently.
- Ensured by:
 - Using **synchronized** methods or blocks.
 - Using **atomic** variables (`AtomicInteger`).
 - Avoiding shared mutable state.
 - Using **concurrent collections** (`ConcurrentHashMap`, etc.)

Pro Tip:

Use `java.util.concurrent` package:

- `ExecutorService` for thread pools
- `CountDownLatch`, `Semaphore`, `CyclicBarrier` for synchronization control
- `ReentrantLock` for advanced locking
- `ConcurrentHashMap` for safe collections

Here's your **Java Concurrency study material** continued with **thread creation, methods, and general concurrency concepts**:

Java Threads & Concurrency Concepts

1. Creating Threads by Extending Thread

You can create a thread by **extending the Thread class** and **overriding the run () method**:



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

public class Main {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        t1.start(); // Starts new thread
    }
}
```

2. Overriding run () Method

- The run () method contains the code that executes when the thread runs.
- You should **never call run () directly** to start a thread.


3. start () VS run ()

Method	Description
start ()	Starts a new thread, invokes run () in a new call stack. 
run ()	Just a normal method call in current thread.  No multithreading

4. Thread Priorities

Java provides **thread priorities** to suggest CPU scheduling order:

```
thread.setPriority(Thread.MIN_PRIORITY); // 1
thread.setPriority(Thread.NORM_PRIORITY); // 5 (default)
thread.setPriority(Thread.MAX_PRIORITY); // 10
```

 Thread priorities may be ignored by some OS/JVMs.

5. Common Methods in Thread Class

 **sleep (long millis)**

Pauses the current thread for given milliseconds.

```
Thread.sleep(1000); // Sleep for 1 second
```

✓ `join()`

Waits for the thread to finish execution.

```
t1.join(); // Main thread waits for t1 to finish
```

✓ `interrupt()`

Interrupts a thread (often used to stop sleeping or waiting threads).

```
t1.interrupt();
```

✓ `isAlive()`

Checks if the thread is still running.

```
if (t1.isAlive()) {  
    System.out.println("Thread is still running.");  
}
```



General Concurrency Concepts (Quick Summary)

- **Race Condition:** Two threads update shared data simultaneously (no sync).
- **Deadlock:** Two or more threads wait forever on each other's locks.
- **Starvation:** A low-priority thread never gets CPU time.
- **Livelock:** Threads keep responding to each other but can't finish.
- **Critical Section:** Code that accesses shared resource.
- **Synchronization:** Mechanism to control thread access to shared resources.
- **Volatile:** Ensures visibility (but not atomicity).
- **Thread Safety:** When class works correctly in a multithreaded environment.

Here's your **Java Concurrency Study Material** (Part 2) focused on **creating threads using Runnable**, along with concepts like **lambdas**, **decoupling**, and **thread reuse**:



Java Threads Using Runnable (Preferred Way)

1. Creating Threads Using Runnable (Preferred Way)

`Runnable` is a **functional interface** with a single method `run()`.

✓ Example:

```
class MyRunnable implements Runnable {  
    public void run() {
```

```
        System.out.println("Runnable thread running: " +
Thread.currentThread().getName());
    }
}

public class Main {
    public static void main(String[] args) {
        Thread t1 = new Thread(new MyRunnable());
        t1.start(); // Starts a new thread
    }
}
```

2. Reusability and Decoupling with Runnable

- Runnable separates **task definition** from **thread execution**.
- You can reuse the same Runnable with multiple threads.
- Promotes **loose coupling**, making code easier to maintain/test.

```
Runnable task = new MyRunnable();
Thread t1 = new Thread(task);
Thread t2 = new Thread(task);
```

3. Anonymous Runnable vs Lambda Runnable

✓ Anonymous Inner Class:

```
Thread t = new Thread(new Runnable() {
    public void run() {
        System.out.println("Anonymous Runnable running");
    }
});
t.start();
```

✓ Lambda Runnable (Java 8+):

```
Runnable r = () -> System.out.println("Lambda Runnable running");
Thread t = new Thread(r);
t.start();
```

Lambdas make the code shorter and more readable.

4. Using Runnable with Thread Class

- Runnable is passed to the Thread constructor.
- Thread handles actual thread creation and scheduling.

```
Runnable task = () -> {
    System.out.println("Task running in thread: " +
Thread.currentThread().getName());
}
```



```
};  
Thread t = new Thread(task);  
t.start();
```

5. One-liner Lambda Runnable

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

This is shorthand for a task that prints "Running" when executed.

6. Executing the Runnable

```
new Thread(r).start();
```

This creates a new thread with the runnable task `r` and starts it.



Summary Table

Feature	Thread (extends)	Runnable (implements)
Inheritance	Uses <code>Thread</code> directly	Better for decoupling
Reusability	Limited	Reusable and testable
Flexibility	Cannot extend other classes	Can implement multiple interfaces
Lambda-friendly	✗	✓ (Java 8+)

Here's a detailed and beginner-friendly explanation of **Java Executors** and why they're a preferred approach for managing threads in concurrent applications:



Java Executors Framework

1. Why Executors Are Better Than Manually Creating Threads



Problems with `new Thread()`:

- Creates a **new thread every time**, which is **expensive**.
- **No control** over the number of threads.
- Manual lifecycle management: hard to `shutdown`, `manage`, or `reuse`.



Executors Solve These with:

► Thread Pooling:

- Reuses a **fixed number of threads** from a pool.
- Greatly reduces overhead of frequent thread creation and destruction.

► Better Resource Management:

- You can **limit** max threads.
- Helps avoid **OutOfMemoryError** and **excessive context switching**.

2. Key Interfaces and Classes

Interface / Class	Description
<code>Executor</code>	Basic interface with <code>execute(Runnable)</code>
<code>ExecutorService</code>	Extends <code>Executor</code> ; adds lifecycle and <code>submit()</code>
<code>ScheduledExecutorService</code>	Allows scheduling tasks to run after delay or periodically
<code>Executors</code> (Utility Class)	Factory for creating thread pools

3. Common Methods in Executor Framework

✓ `submit(Callable/Runnable task)`

- Submits a task for execution and returns a `Future`.

```
Future<String> future = executorService.submit(() -> "Task Completed");
```

✓ `execute(Runnable task)`

- Executes the task, but **does not return a result**.

```
executor.execute(() -> System.out.println("Executed"));
```

✓ `shutdown()`

- Initiates an **orderly shutdown**.
- Tasks already submitted will execute, no new ones accepted.

```
executor.shutdown();
```

✓ `shutdownNow()`

- **Forces shutdown** immediately and attempts to stop all active tasks.

```
executor.shutdownNow();
```

✅ **invokeAll** (Collection<Callable>)

- Runs all tasks and waits for **all to finish**, returns list of `Future`.

```
List<Callable<String>> tasks = List.of(() -> "One", () -> "Two");
List<Future<String>> results = executor.invokeAll(tasks);
```

✅ **invokeAny** (Collection<Callable>)

- Runs all tasks, **returns result of the first completed**, cancels the rest.

```
String firstResult = executor.invokeAny(tasks);
```

✅ **Example:**

```
import java.util.concurrent.*;

public class ExecutorExample {
    public static void main(String[] args) throws Exception {
        ExecutorService executor = Executors.newFixedThreadPool(2);

        Runnable task = () -> System.out.println("Running task " +
            Thread.currentThread().getName());
        executor.execute(task);

        Future<String> future = executor.submit(() -> "Callable result");

        System.out.println("Result: " + future.get());

        executor.shutdown();
    }
}
```

Summary

Feature	Thread	Executor
Thread Creation	Manual	Automatic (via pool)
Resource Mgmt	Poor	Excellent
Scalability	Limited	Scalable
Code Simplicity	Verbose	Cleaner with API support

Java Concurrency – Advanced Concepts

4. Callable vs Runnable

Feature	Runnable	Callable<V>
Return value	No	Yes (returns a value of type v)
Throws Exception	No (must handle checked exceptions)	Yes (can throw checked exceptions)
Method	public void run()	public V call() throws Exception
Use case	When result is not needed	When you want a result or may throw

Example:

```
Callable<String> callableTask = () -> {
    Thread.sleep(1000);
    return "Callable Result";
};

Runnable runnableTask = () -> System.out.println("Runnable Running");
```

5. Future and FutureTask

Future<V>:

Represents the result of an asynchronous computation.

- `get()`: Waits if necessary and returns the result.
- `isDone()`: Checks if the task is completed.
- `cancel(true/false)`: Attempts to cancel execution.

```
ExecutorService executor = Executors.newSingleThreadExecutor();
Future<String> future = executor.submit(callableTask);

System.out.println(future.isDone()); // false
String result = future.get();         // blocks until complete
System.out.println(result);           // Callable Result
executor.shutdown();
```

FutureTask<V>:

Implements `Runnable` and `Future` — you can run it with a thread or submit to an executor.

```
FutureTask<String> futureTask = new FutureTask<>(callableTask);
new Thread(futureTask).start();
System.out.println(futureTask.get());
```

6. ThreadPool Types

Java provides convenient thread pool creation via `Executors` factory methods.

✓ `newFixedThreadPool(int nThreads)`

- Fixed number of threads.
- Good for stable workloads.

```
ExecutorService pool = Executors.newFixedThreadPool(4);
```

✓ `newCachedThreadPool()`

- Creates new threads as needed and reuses idle ones.
- Good for many short-lived tasks.

```
ExecutorService pool = Executors.newCachedThreadPool();
```

✓ `newSingleThreadExecutor()`

- Single thread for all tasks (executes sequentially).
- Useful when tasks must not overlap.

```
ExecutorService pool = Executors.newSingleThreadExecutor();
```

✓ `newScheduledThreadPool(int corePoolSize)`

- Supports scheduled and periodic task execution.

```
ScheduledExecutorService scheduler = Executors.newScheduledThreadPool(2);
scheduler.schedule(() -> System.out.println("Delayed Task"), 2,
    TimeUnit.SECONDS);
```

Summary Table

Feature	Purpose
Callable	Returns result and can throw checked exceptions

Feature	Purpose
<code>Runnable</code>	No result, no checked exceptions
<code>Future.get()</code>	Blocks and returns result
<code>Future.isDone()</code>	Checks if task is complete
<code>Future.cancel()</code>	Attempts to cancel the task
<code>newFixedThreadPool</code>	Fixed number of threads
<code>newCachedThreadPool</code>	Dynamically created threads (as needed)
<code>newSingleThreadExecutor</code>	One thread for all tasks
<code>newScheduledThreadPool</code>	For delayed and periodic tasks

JAVA COLLECTIONS & UTILITIES – DETAILED STUDY GUIDE

1. Collections Framework Core Interfaces

✓ Introduction

Java Collections Framework (JCF) is a set of classes and interfaces that implement commonly reusable collection data structures like lists, sets, and maps.

◆ Core Interfaces

Interface	Ordered	Allows Duplicates	Key-Value?	Example Classes
List	✓ Yes	✓ Yes	✗ No	ArrayList, LinkedList
Set	✗ No	✗ No	✗ No	HashSet, TreeSet
Map	✗ No	✓ Yes (values)	✓ Yes	HashMap, TreeMap
Queue	✓ Yes	✓ Yes	✗ No	LinkedList, PriorityQueue

2. Common Implementations

◆ List Implementations

1. ArrayList

- Backed by **dynamic array**.
- **Fast random access**, slow insert/delete in the middle.
- Resizable automatically.

```
List<String> names = new ArrayList<>();
names.add("Alice");
names.add("Bob");
```

2. LinkedList

- Backed by **doubly linked list**.
- **Fast insert/delete** from anywhere.

- Slower access by index.

```
List<Integer> numbers = new LinkedList<>();  
numbers.add(1);  
numbers.add(2);
```

◆ Set Implementations

1. HashSet

- Unordered.
- No duplicates.
- Uses `hashCode()` and `equals()`.

```
Set<String> set = new HashSet<>();  
set.add("apple");  
set.add("banana");
```

2. LinkedHashSet

- Maintains insertion order.
- No duplicates.

```
Set<String> orderedSet = new LinkedHashSet<>();  
orderedSet.add("apple");  
orderedSet.add("banana");
```

3. TreeSet

- Sorted using natural/comparator order.
- No duplicates.

```
Set<Integer> sortedSet = new TreeSet<>();  
sortedSet.add(5);  
sortedSet.add(2);
```

◆ Map Implementations

1. HashMap

- Stores key-value pairs.
- Unordered.
- Allows one `null` key and multiple `null` values.

```
Map<String, Integer> map = new HashMap<>();  
map.put("Alice", 90);  
map.put("Bob", 80);
```


2. LinkedHashMap

- Maintains insertion order.
- Useful for caches.

```
Map<String, String> lhm = new LinkedHashMap<>();  
lhm.put("one", "first");  
lhm.put("two", "second");
```

3. TreeMap

- Sorted by keys.
- No null keys allowed.

```
Map<String, String> tm = new TreeMap<>();  
tm.put("c", "Charlie");  
tm.put("a", "Alice");
```

◆ Queue Implementation

1. PriorityQueue

- Ordered by natural order or comparator.
- Default: Min-heap (smallest element at head).

```
Queue<Integer> pq = new PriorityQueue<>();  
pq.add(5);  
pq.add(1);  
pq.add(3);
```

3. Utility Classes

◆ Collections Class (java.util.Collections)

Method	Description
<code>sort(List)</code>	Sorts in natural order
<code>reverse(List)</code>	Reverses the list
<code>shuffle(List)</code>	Randomly shuffles
<code>max(Collection)</code>	Max element
<code>min(Collection)</code>	Min element
<code>unmodifiableList()</code>	Returns read-only version

```
List<Integer> nums = Arrays.asList(3, 1, 2);
Collections.sort(nums);
Collections.reverse(nums);
```

◆ Arrays Class (java.util.Arrays)

Method	Description
<code>sort(array)</code>	Sorts array
<code>binarySearch(array, key)</code>	Searches sorted array
<code>equals(arr1, arr2)</code>	Compares arrays
<code>copyOf(array, newLength)</code>	Copies array to new size

```
int[] a = {5, 2, 3};
Arrays.sort(a);
int idx = Arrays.binarySearch(a, 3);
```

◆ Objects Class (java.util.Objects)

Method	Description
<code>equals(a, b)</code>	Null-safe equality check
<code>hashCode(a)</code>	Null-safe hashCode
<code>requireNonNull(a)</code>	Throws NPE if null

```
Objects.requireNonNull(name, "Name cannot be null");
```

4. Date and Time Utilities

◆ Legacy Classes

✓ Date

- Stores date and time.
- Deprecated in favor of `java.time`.

```
Date date = new Date();
System.out.println(date);
```

✓ Calendar

- Abstract class with `GregorianCalendar` implementation.
- Allows field-wise manipulation.

```
Calendar cal = Calendar.getInstance();
cal.set(Calendar.YEAR, 2025);
Date updatedDate = cal.getTime();
```

✓ TimeZone

- Used with `Calendar` for international dates.

```
TimeZone tz = TimeZone.getTimeZone("Asia/Kolkata");
Calendar cal = Calendar.getInstance(tz);
```

✓ Locale

- For localization and formatting.

```
Locale locale = new Locale("fr", "FR");
DateFormat df = DateFormat.getDateInstance(DateFormat.LONG, locale);
System.out.println(df.format(new Date()));
```

◆ Modern Java 8+ – `java.time`

Class	Purpose
<code>LocalDate</code>	Only date
<code>LocalTime</code>	Only time
<code>LocalDateTime</code>	Date + Time
<code>ZonedDateTime</code>	Includes timezone
<code>DateTimeFormatter</code>	For formatting
<code>Duration</code> , <code>Period</code>	Time difference

```
LocalDate date = LocalDate.now();
LocalTime time = LocalTime.now();
LocalDateTime dt = LocalDateTime.of(date, time);

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

✓ Date Difference Example:

```
LocalDate start = LocalDate.of(2020, 1, 1);
LocalDate end = LocalDate.now();
Period period = Period.between(start, end);
System.out.println("Years: " + period.getYears());
```

Here's the **detailed study material** for the remaining two sections from your list:

JAVA COLLECTIONS & UTILITIES – PART 2

5. Other Useful Classes

Random – Pseudo-Random Number Generator

- Belongs to `java.util`.
- Uses a seed internally; produces the same sequence if same seed is used.
- Useful for games, simulations, etc.

```
import java.util.Random;

Random rand = new Random();
int randomInt = rand.nextInt(100); // 0 to 99
double randomDouble = rand.nextDouble(); // 0.0 to 1.0
boolean randomBool = rand.nextBoolean();
```

Scanner – Input Handling

- Reads from `System.in`, files, strings, etc.
- Belongs to `java.util`.

```
import java.util.Scanner;

Scanner sc = new Scanner(System.in);
System.out.print("Enter name: ");
String name = sc.nextLine();
System.out.print("Enter age: ");
int age = sc.nextInt();
sc.close();
```

✓ Also supports reading from **files**:

```
Scanner fileScanner = new Scanner(new File("data.txt"));
while (fileScanner.hasNextLine()) {
```

```
        System.out.println(fileScanner.nextLine());
    }
}
```

◆ Properties – Key-Value Configuration

- Used to store application settings (e.g., `config.properties`).
- Based on `Hashtable`.

```
import java.util.Properties;
import java.io.*;

Properties prop = new Properties();
prop.load(new FileInputStream("config.properties"));
System.out.println(prop.getProperty("username"));

prop.setProperty("theme", "dark");
prop.store(new FileOutputStream("config.properties"), "Updated theme");
```

◆ Timer and TimerTask – Scheduling Tasks

- `Timer`: schedules a task for future execution.
- `TimerTask`: represents the task to run.

```
import java.util.*;

TimerTask task = new TimerTask() {
    public void run() {
        System.out.println("Task executed!");
    }
};

Timer timer = new Timer();
timer.schedule(task, 3000); // Runs once after 3 seconds
```

✓ You can also schedule repeating tasks:

```
timer.scheduleAtFixedRate(task, 0, 1000); // every 1 second
```

6. Legacy Collections

These are **thread-safe but outdated**. Use modern equivalents from the Collections Framework unless synchronization is needed.

◆ Vector

- Like `ArrayList` but synchronized (thread-safe).
- Slower than `ArrayList` in single-threaded apps.

```
Vector<String> vector = new Vector<>();
vector.add("Hello");
vector.add("World");
```

◆ Stack (Extends Vector)

- LIFO (Last In First Out) structure.
- Use Deque (like ArrayDeque) instead in modern code.

```
Stack<Integer> stack = new Stack<>();
stack.push(10);
stack.push(20);
System.out.println(stack.pop()); // 20
```

◆ Hashtable

- Like HashMap but synchronized.
- Doesn't allow null keys or null values.
- Replaced by ConcurrentHashMap for thread-safe needs.

```
Hashtable<String, String> table = new Hashtable<>();
table.put("username", "admin");
table.put("password", "1234");
```

✓ Modern Alternatives Summary:

Legacy Class	Modern Alternative
Vector	ArrayList
Stack	Deque (ArrayDeque)
Hashtable	HashMap / ConcurrentHashMap

✓ BONUS TIP: When to Use Synchronized Collections

If you need **thread-safe** collections:

```
List<String> syncList = Collections.synchronizedList(new ArrayList<>());
Map<String, String> syncMap = Collections.synchronizedMap(new HashMap<>());
```

Or use `java.util.concurrent` classes like:

- ConcurrentHashMap
- CopyOnWriteArrayList
- BlockingQueue

Here is the **detailed study material** for the final sections of your Java topics list:

JAVA COLLECTIONS & UTILITIES – PART 3

7. Enumerations

◆ Enumeration Interface (Legacy)

- Found in `java.util`.
- Used mainly with **legacy classes** like `Vector` and `Hashtable`.
- Works similar to an `Iterator`, but only has two methods:

- `hasMoreElements()`
- `nextElement()`

▼ Example:

```
import java.util.*;

Vector<String> vector = new Vector<>();
vector.add("Java");
vector.add("Python");

Enumeration<String> e = vector.elements();
while (e.hasMoreElements()) {
    System.out.println(e.nextElement());
}
```

Notes:

- No `remove()` method like `Iterator`.
 - For modern code, use `Iterator` instead.
-

8. Wrapper Classes & Conversions

◆ Autoboxing / Unboxing

- **Autoboxing**: Converts **primitive** → **wrapper class** automatically.
- **Unboxing**: Converts **wrapper** → **primitive** automatically.

▼ Example:

```
Integer x = 10;           // autoboxing
int y = x + 5;            // unboxing
```

Primitive	Wrapper
int	Integer
char	Character
double	Double
boolean	Boolean
...	...

◆ Arrays ↔ Collections Conversion

✓ Array → List

```
String[] arr = {"A", "B", "C"};
List<String> list = Arrays.asList(arr); // Fixed-size list
```

● Warning: `Arrays.asList()` returns a **fixed-size** list (can't add/remove items).

To make it modifiable:

```
List<String> modifiableList = new ArrayList<>(Arrays.asList(arr));
```

✓ List → Array

```
List<String> list = new ArrayList<>();
list.add("X");
list.add("Y");

String[] array = list.toArray(new String[0]);
```

9. Comparable & Comparator

◆ Comparable<T> – Natural Ordering

- Interface in `java.lang`
- Implemented by classes to define **default sorting** behavior.

▣ Example:


```
class Student implements Comparable<Student> {
    int id;
    String name;

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

    public int compareTo(Student other) {
        return this.id - other.id; // Sort by id
    }
}
```

Used with:

```
Collections.sort(studentList); // Uses compareTo()
```

◆ Comparator<T> – Custom Sorting

- Interface in `java.util`
- Used to define **external sorting logic**.

▼ Example: Sort `Student` by name:

```
Comparator<Student> nameComparator = new Comparator<Student>() {
    public int compare(Student s1, Student s2) {
        return s1.name.compareTo(s2.name);
    }
};
Collections.sort(studentList, nameComparator);
```

NEW With Lambda:

```
Collections.sort(studentList, (a, b) -> a.name.compareTo(b.name));
```

◆ TreeSet / TreeMap with Custom Comparator

```
Set<String> treeSet = new TreeSet<>((a, b) -> b.compareTo(a)); // Reverse order
```

✅ Summary Table: Comparable vs Comparator

Feature	Comparable	Comparator
Package	<code>java.lang</code>	<code>java.util</code>

Feature	Comparable	Comparator
Method	<code>compareTo(T o)</code>	<code>compare(T o1, T o2)</code>
Used for	Natural/default sorting	Custom sorting
Modifies class?	Yes, must implement interface	No, separate from the class
Example use	<code>Collections.sort(list)</code>	<code>Collections.sort(list, comparator)</code>

TOPIC 14 DATABASE PROGRAMMING

1. JDBC Driver Loading

Driver Loading (Old Way – Pre JDBC 4.0)

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

- Dynamically loads the MySQL JDBC driver class into memory.
- Needed before JDBC 4.0.

From JDBC 4.0+

- Driver loading is **automatic** if the driver JAR is in the classpath.
- Uses the **Service Provider Mechanism** (META-INF/services).

JDBC URL Format

Example (MySQL):

```
jdbc:mysql://localhost:3306/mydatabase
```

- **jdbc:mysql** – protocol and subprotocol
- **localhost:3306** – host and port
- **mydatabase** – database name

2. Types of JDBC Drivers

Type	Description	Pros	Cons
Type 1	JDBC-ODBC Bridge	Quick for testing	Deprecated, not platform-independent
Type 2	Native API Driver	Fast, native access	Needs native libraries – platform dependent
Type 3	Network Protocol Driver	Flexible with middleware	Slower due to middleware
Type 4	Thin Driver (Pure Java)	Best portability and performance	Needs correct DB driver for each DB

 **Recommendation:** Use **Type 4 Driver** in modern apps.

3. Connection Interface

✓ Getting a Connection

```
Connection conn = DriverManager.getConnection(
    "jdbc:mysql://localhost:3306/mydb", "username", "password");
```

✓ Auto-commit Mode

- **Default:** `conn.setAutoCommit(true);`
- Every SQL statement is committed automatically.

✓ Manual Transaction

```
conn.setAutoCommit(false);
try {
    // execute SQL statements
    conn.commit(); // commit transaction
} catch (SQLException e) {
    conn.rollback(); // rollback on error
}
```

✓ Always Close Resources

```
if (conn != null) {
    conn.close(); // Prevents memory leaks
}
```

Summary

Topic	Key Point
Driver Loading	Use <code>Class.forName()</code> pre-JDBC 4.0. Automatic since JDBC 4.0.
JDBC URL	Must follow correct syntax: <code>jdbc:subprotocol://host:port/dbname</code>
Driver Types	Type 4 (Thin driver) is the best for modern Java apps
Connection	Use <code>DriverManager.getConnection()</code> , manage transactions carefully

JDBC STUDY MATERIAL

1. JDBC Driver Loading

- **Old Way (Pre JDBC 4.0):**

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

This loads the JDBC driver class.

- **Modern Way (JDBC 4.0+):**
 - Automatic loading via Service Provider mechanism if driver JAR is in classpath.
- **JDBC URL Format:**

```
jdbc:mysql://host:port/dbname
```

2. Types of JDBC Drivers

- **Type 1: JDBC-ODBC Bridge** (Deprecated)
- **Type 2: Native API Driver** (Partially Java, uses native code)
- **Type 3: Network Protocol Driver** (Middleware-based)
- **Type 4: Thin Driver** (Pure Java, widely used)

Summary:

Type	Pure Java	Needs Native Libs	Suitable for Web	Performance
1	✗	✓	✗	✗
2	✗	✓	✗	⚠
3	⚠	✗	⚠	⚠
4	✓	✗	✓	✓

3. Connection Interface

- Obtained using:

```
Connection conn = DriverManager.getConnection(url, user, password);
```

- **Auto-commit** is `true` by default.
- Set `false` for manual control:

```
conn.setAutoCommit(false);
```

- Always close connections to avoid resource leaks.
-

4. Statement Interface

- Used to execute **static** SQL.
- Example:

```
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("SELECT * FROM users");
int count = stmt.executeUpdate("UPDATE users SET score = 10");
```

5. PreparedStatement Interface

- Used for **parameterized** queries.
- Helps prevent **SQL Injection**.
- Better performance due to **pre-compilation**.
- Example:

```
PreparedStatement pstmt = conn.prepareStatement("SELECT * FROM users WHERE id = ?");
pstmt.setInt(1, 101);
ResultSet rs = pstmt.executeQuery();
```

6. ResultSet Interface

- Holds query results.
- Common methods:
 - `next()`, `getInt()`, `getString()`, `getDate()`
 - `wasNull()` – check if last read column was NULL

Types:

- `TYPE_FORWARD_ONLY` – default, one-way
- `TYPE_SCROLL_INSENSITIVE` – scrollable, not sensitive to DB changes
- `TYPE_SCROLL_SENSITIVE` – reflects DB changes while scrolling

Concurrency:

- `CONCUR_READ_ONLY` – can't update
 - `CONCUR_UPDATABLE` – can update result
-

7. CallableStatement Interface

- Used to **call stored procedures** in DB.
- Supports **IN, OUT, INOUT** parameters.

Example:

```
CallableStatement cstmt = conn.prepareCall("{call getEmployeeName(?, ?)}");
cstmt.setInt(1, 1001); // IN parameter
cstmt.registerOutParameter(2, Types.VARCHAR); // OUT parameter
cstmt.execute();
String name = cstmt.getString(2);
```

8. Connection Pooling

- Improves performance by reusing DB connections.
- Avoids overhead of creating/closing connections repeatedly.

Popular Libraries:

- **HikariCP** (fastest & modern)
- **Apache DBCP**
- **C3P0**

Using DataSource:

```
DataSource ds = ... // configured pool
Connection conn = ds.getConnection();
```

- Connections from a pool are returned instead of created anew.

JDBC STUDY MATERIAL

1. JDBC Driver Loading

- **Old Way (Pre JDBC 4.0):**

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

This loads the JDBC driver class.

- **Modern Way (JDBC 4.0+):**
 - Automatic loading via Service Provider mechanism if driver JAR is in classpath.
- **JDBC URL Format:**

```
jdbc:mysql://host:port/dbname
```

2. Types of JDBC Drivers

- **Type 1: JDBC-ODBC Bridge** (Deprecated)
- **Type 2: Native API Driver** (Partially Java, uses native code)

- **Type 3: Network Protocol Driver** (Middleware-based)
- **Type 4: Thin Driver** (Pure Java, widely used)

Summary:

Type	Pure Java	Needs Native Libs	Suitable for Web	Performance
1	✗	✓	✗	✗
2	✗	✓	✗	⚠
3	⚠	✗	⚠	⚠
4	✓	✗	✓	✓

3. Connection Interface

- Obtained using:

```
Connection conn = DriverManager.getConnection(url, user, password);
```

- **Auto-commit** is `true` by default.
- Set `false` for manual control:

```
conn.setAutoCommit(false);
```

- Always close connections to avoid resource leaks.

4. Statement Interface

- Used to execute **static SQL**.
- Example:

```
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("SELECT * FROM users");
int count = stmt.executeUpdate("UPDATE users SET score = 10");
```

5. PreparedStatement Interface

- Used for **parameterized** queries.
- Helps prevent **SQL Injection**.
- Better performance due to **pre-compilation**.
- Example:


```
PreparedStatement pstmt = conn.prepareStatement("SELECT * FROM users  
WHERE id = ?");  
pstmt.setInt(1, 101);  
ResultSet rs = pstmt.executeQuery();
```

6. ResultSet Interface

- Holds query results.
- Common methods:
 - `next()`, `getInt()`, `getString()`, `getDate()`
 - `wasNull()` – check if last read column was NULL

Types:

- `TYPE_FORWARD_ONLY` – default, one-way
- `TYPE_SCROLL_INSENSITIVE` – scrollable, not sensitive to DB changes
- `TYPE_SCROLL_SENSITIVE` – reflects DB changes while scrolling

Concurrency:

- `CONCUR_READ_ONLY` – can't update
- `CONCUR_UPDATABLE` – can update result

7. CallableStatement Interface

- Used to **call stored procedures** in DB.
- Supports **IN**, **OUT**, **INOUT** parameters.

Example:

```
CallableStatement cstmt = conn.prepareCall("{call getEmployeeName(?, ?)}");  
cstmt.setInt(1, 1001); // IN parameter  
cstmt.registerOutParameter(2, Types.VARCHAR); // OUT parameter  
cstmt.execute();  
String name = cstmt.getString(2);
```

8. Connection Pooling

- Improves performance by reusing DB connections.
- Avoids overhead of creating/closing connections repeatedly.

Popular Libraries:

- **HikariCP** (fastest & modern)
- **Apache DBCP**
- **C3P0**

Using DataSource:

```
DataSource ds = ... // configured pool  
Connection conn = ds.getConnection();
```

- Connections from a pool are returned instead of created anew.
-

◆ What is ResultSet?

`ResultSet` is an **interface** in **JDBC** that represents the **result of a database query**—typically a `SELECT` statement.

💡 Think of it like a table in memory:

- Rows = records returned by the query
- Columns = fields in each record

You can **move through rows** and **read column values** using `ResultSet` methods.

◆ Important Methods:

✅ `next()`

- Moves the cursor to the **next row**.
- Returns `true` if there's a row, `false` if you've reached the end.

```
while (rs.next()) {  
    // Process row  
}
```

✅ `getString()`, `getInt()`, `getDate()`, etc.

- Retrieves the value from a **column** in the current row.
- You can access by **column name** or **column index** (starting at 1).

```
String name = rs.getString("username");  
int age = rs.getInt(2);
```

✅ `wasNull()`

- Checks if the **last value read** was `SQL NULL`.

```
int age = rs.getInt("age");  
if (rs.wasNull()) {  
    // age was actually NULL in the DB  
}
```

◆ ResultSet Types (Scrollability)

This affects how you can **navigate through rows** in the `ResultSet`.

Type	Description
TYPE_FORWARD_ONLY	(Default) Only move forward using <code>next()</code>
TYPE_SCROLL_INSENSITIVE	Scrollable both ways; doesn't reflect DB changes after the <code>ResultSet</code> is created
TYPE_SCROLL_SENSITIVE	Scrollable and reflects DB changes dynamically

Example of scrollable `ResultSet`:

```
Statement stmt = conn.createStatement(
    ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_READ_ONLY);
ResultSet rs = stmt.executeQuery("SELECT * FROM users");

rs.last(); // Move to last row
int total = rs.getRow(); // Get row number
rs.beforeFirst(); // Move to before the first row
```

◆ Concurrency (Updatability)

This controls whether you can **modify the data** in the `ResultSet`.

Concurrency	Description
CONCUR_READ_ONLY	You can only read data
CONCUR_UPDATABLE	You can update, insert, or delete rows through the <code>ResultSet</code>

Example of updatable `ResultSet`:

```
Statement stmt = conn.createStatement(
    ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_UPDATABLE);
ResultSet rs = stmt.executeQuery("SELECT * FROM users");

if (rs.next()) {
    rs.updateString("username", "newName");
    rs.updateRow(); // Commit change to DB
}
```

✓ Summary

Feature	Purpose
<code>next()</code>	Move to next row
<code>getXxx()</code>	Get value of a column

Feature	Purpose
<code>wasNull()</code>	Check if last column was SQL NULL
<code>TYPE_*</code>	Scroll behavior of the ResultSet
<code>CONCUR_*</code>	Whether you can update the ResultSet

Let me know if you'd like diagrams or examples with actual output.

◆ What is Connection Pooling?

Connection Pooling is a technique to **improve performance** by **reusing database connections** instead of creating and closing them again and again.

! Why is this important?

- Opening a new DB connection is **slow** (network, authentication, setup).
 - In high-traffic apps (like web apps), creating new connections repeatedly causes **high overhead**.
 - Connection pools solve this by:
 - Creating a **"pool" of ready-to-use connections**
 - Giving one when needed
 - Returning it to the pool when done
-

◆ How it works:

1. On app startup: Pool creates, say, 10 DB connections.
 2. When you need one: You get it from the pool (via `dataSource.getConnection()`).
 3. When done: You **don't close it** — you **return it to the pool**.
-

◆ Key Components:

✓ `javax.sql.DataSource`





- Interface for managing pooled connections.
- Used instead of `DriverManager`.
- You get a connection like this:

```
Connection conn = dataSource.getConnection();
```

◆ Popular Connection Pool Libraries:

Library	Notes
HikariCP	Fastest, lightweight, commonly used (default in Spring Boot)
Apache DBCP	Stable, older option
C3P0	Easy to configure, used in Hibernate apps

◆ Benefits of Connection Pooling:

Benefit	Explanation
 Fast	Reuses existing connections — no delay
 Scalable	Handles many users more efficiently
 Configurable	You can set max/min connections, timeout, etc.
 Cleaner	Connections are managed centrally by the pool

✅ Code Example (HikariCP)

```
HikariConfig config = new HikariConfig();
config.setJdbcUrl("jdbc:mysql://localhost:3306/mydb");
config.setUsername("root");
config.setPassword("password");
HikariDataSource dataSource = new HikariDataSource(config);

Connection conn = dataSource.getConnection();
// Use the connection...
conn.close(); // returns to pool, not actually closed
```

← END Summary

Concept	Description
Connection Pooling	Reuses DB connections
DataSource	Interface to get pooled connections
getConnection()	Fetches a connection from the pool
Libraries	HikariCP, DBCP, C3P0
Main Benefit	Faster, efficient, scalable DB access

TOPIC 19 Database Indexes

1. Definition

An **index** is a specialized data structure that database engines maintain to speed up data retrieval operations. Conceptually, an index is analogous to the index in a book: rather than scanning every page to find a topic, you consult the index to jump directly to the relevant page.

- **Core idea:** Store a sorted reference (pointer) to the rows in a table so lookups by key(s) can be done in $O(\log n)$ time instead of $O(n)$.
- **Physical structures:**
 - **B-Tree / B+-Tree:** The most common implementation, balances read/write performance.
 - **Hash index:** Excellent for equality lookups ($=$), but not for range scans ($>$, $<$).
 - **Bitmap index:** Efficient for low-cardinality columns (few distinct values).

2. Types of Indexes

2.1 By Column Coverage

Type	Definition
Single-column	Index on one column.
Composite (multi-column)	Index on two or more columns. The order of columns matters (e.g. (A, B) can be used for queries on A alone, but not on B alone).

```
-- Single-column index
CREATE INDEX idx_last_name
  ON employees(last_name);

-- Composite index on (department_id, salary)
CREATE INDEX idx_dept_salary
  ON employees(department_id, salary);
```

2.2 By Uniqueness

Type	Definition
Unique	Enforces that no two rows have the same indexed key value(s).
Non-unique	Allows duplicate key values; speeds up lookups without enforcing uniqueness.


```
-- Unique index: prevents duplicate email addresses
CREATE UNIQUE INDEX idx_unique_email
ON employees(email);
```

2.3 By Physical Organization

Type	Definition
Clustered	The table's rows are stored on disk in the same order as the index. A table can have <i>one</i> clustered index only.
Non-clustered	The index is stored separately from the table data; contains pointers (row IDs) to the actual table rows.

- **Clustered index** is typically the primary key. Changes physical ordering.
- **Non-clustered index** does not affect row order, but builds a separate lookup structure.

3. Benefits

1. Speeds up SELECT queries

- Index seeks replace full table scans for equality/range predicates.
- Example:

```
EXPLAIN SELECT * FROM employees WHERE last_name = 'Patel';
-- Uses idx_last_name for an index seek
```

2. Improves JOIN performance

- Index on join columns allows efficient nested-loop or merge joins.

3. Enhances WHERE, ORDER BY, and GROUP BY

- Queries with filtering, sorting, or grouping predicates can leverage indexes to avoid sorting/aggregation on the entire table.
- Example:

```
SELECT department_id, COUNT(*)
FROM employees
GROUP BY department_id;
-- If idx_dept exists, grouping can scan the index in sorted order
```

4. Drawbacks

1. Slower INSERT, UPDATE, and DELETE

- Each DML operation must update all relevant indexes.
- Impact increases with the number of indexes on a table.

2. Additional Storage Overhead

- Each index consumes disk space (and memory when cached).
- Composite and clustered indexes tend to be larger.

3. Maintenance Complexity

- Over-indexing can lead to diminishing returns—too many indexes hurt write throughput without proportional read gains.

5. When to Use Indexes

Scenario	Recommendation
Columns frequently used in <code>WHERE</code> clauses	Create a single-column index for equality/range filters.
Columns used in <code>JOIN</code> conditions	Index both sides of the join on their join-key columns.
Columns in <code>ORDER BY</code> or <code>GROUP BY</code>	Composite index matching the sort/group key order.
High-cardinality columns (many distinct values)	Good candidates for non-unique indexes.
Low-cardinality columns (few distinct values)	Consider bitmap indexes (if supported) or skip indexing.
Tables with heavy write activity	Be judicious: limit indexes to only those that deliver real benefit.

6. Examples & Best Practices

6.1 Example: Optimizing a Query

Unoptimized (full scan):

```
SELECT *  
FROM orders  
WHERE customer_id = 12345;
```

Explain plan shows full table scan → add index:

```
CREATE INDEX idx_orders_customer  
ON orders(customer_id);
```

After indexing:

```
-- Now performs an index seek on idx_orders_customer
EXPLAIN SELECT *
FROM orders
WHERE customer_id = 12345;
```

6.2 Covering Index

When an index includes *all* columns needed by a query, the database can satisfy the query *entirely* from the index (no table lookup needed).

```
-- Covering index for this query:
CREATE INDEX idx_orders_covering
  ON orders(customer_id, order_date, total_amount);

SELECT order_date, total_amount
FROM orders
WHERE customer_id = 12345;
```

6.3 Monitoring & Maintenance

- **Use EXPLAIN/EXPLAIN ANALYZE** to verify index usage.
- **Rebuild/Reorganize** indexes periodically (depending on fragmentation).
- **Drop unused indexes** to reclaim space and speed up writes.
- **Limit composite indexes** to essential combinations.

7. Summary

- **Indexes** are pivotal for high-performance read operations but carry write and storage costs.
- Choose the right **type** (single vs composite, unique vs non-unique, clustered vs non-clustered) based on query patterns.
- Follow best practices: monitor usage, avoid over-indexing, and maintain indexes regularly.

In-Depth Study Material: Primary Keys & Foreign Keys

Primary Key

1. Definition

A **primary key** is a column—or a set of columns—that **uniquely** identifies each row in a database table. It ensures that no two rows share the same key value.

2. Properties

- **Uniqueness:** Every value in the primary key column(s) must be unique across the table.
- **Non-nullable:** Primary key columns cannot contain `NULL`.
- **Singularity:** A table can define **exactly one** primary key constraint.

- **Immutability (Best Practice):** Key values should rarely change once assigned.

3. Automatic Indexing

Defining a primary key automatically creates a **unique index** on the key column(s), enabling efficient lookups and enforcement of uniqueness.

```
-- Single-column primary key
CREATE TABLE departments (
  department_id INT PRIMARY KEY,
  department_name VARCHAR(100) NOT NULL
);

-- Composite (multi-column) primary key
CREATE TABLE employee_projects (
  employee_id INT,
  project_id INT,
  PRIMARY KEY (employee_id, project_id)
);
```

4. Types of Primary Keys

- **Surrogate Key:** An artificially generated identifier (e.g., auto-increment integer or UUID).
- **Natural Key:** An existing real-world attribute (e.g., Social Security Number, email address).

Pros & Cons

Type	Pros	Cons
Surrogate	Simplicity, no business logic changes	Less meaningful data, extra storage
Natural	Meaningful, no extra column required	Can change over time, risk of violating immutability

5. Best Practices

- Prefer **surrogate keys** for stability in large, evolving schemas.
- Name constraints explicitly:

```
CONSTRAINT pk_departments PRIMARY KEY (department_id)
```

- Avoid using volatile business data (e.g., email) as natural keys if values may change.

Foreign Key

1. Definition

A **foreign key** is a column (or set of columns) in one table (the *child*) that **references** the primary key or a unique key in another table (the *parent*), establishing a relational link.

2. Purpose

- **Referential Integrity:** Ensures that every value in the child table’s foreign key column exists in the parent table.
- **Data Consistency:** Prevents orphaned records and enforces valid relationships.

3. Constraints

- **Existence:** Foreign key values must match existing primary/unique key values in the parent table (or be NULL if the FK is defined as nullable).
- **Single Definition:** A child table can have multiple foreign keys to various parent tables, but each FK constraint is defined separately.

```
CREATE TABLE employees (  
  employee_id INT PRIMARY KEY,  
  first_name VARCHAR(50),  
  department_id INT,  
  CONSTRAINT fk_emp_dept FOREIGN KEY (department_id)  
    REFERENCES departments(department_id)  
);
```

4. Referential Actions (ON DELETE / ON UPDATE)

When the parent key is updated or deleted, the database must know how to propagate changes. Common actions:

Action	Effect on Child Rows
CASCADE	Automatically delete or update child rows to match the parent change.
SET NULL	Set the foreign key in child rows to NULL (only if the FK column allows NULL).
RESTRICT	Prevent deletion/update on the parent if any matching child rows exist.
NO ACTION	Same as RESTRICT in most systems: disallow the change until child rows are removed.
SET DEFAULT	Set the FK column to its default value (if a default is defined).

```
CREATE TABLE orders (  
  order_id INT PRIMARY KEY,  
  customer_id INT,  
  CONSTRAINT fk_orders_customer FOREIGN KEY (customer_id)  
    REFERENCES customers(customer_id)  
    ON DELETE CASCADE  
    ON UPDATE NO ACTION  
);
```

5. Composite Foreign Keys

Foreign keys can span multiple columns to match a composite primary (or unique) key in the parent:

```
CREATE TABLE assignments (  
  employee_id INT,  
  project_id INT,  
  role VARCHAR(50),  
  PRIMARY KEY (employee_id, project_id),  
  CONSTRAINT fk_assign_proj  
    FOREIGN KEY (project_id)  
    REFERENCES projects(project_id)  
);
```

6. Indexing Foreign Keys

- Many databases do **not** automatically index foreign key columns.
- **Best practice:** Manually create an index on the FK column if you frequently join child to parent tables:

```
CREATE INDEX idx_emp_dept_fk  
ON employees(department_id);
```

7. Comparison & Summary

Aspect	Primary Key	Foreign Key
Role	Uniquely identifies rows in its own table	Creates link to another table's primary/unique key
Uniqueness	Enforced (unique + non-null)	Not necessarily unique; enforces existence only
Nullability	Disallowed (NOT NULL)	Allowed if defined as nullable
Automatic Index	Yes (unique index)	No (index must be added manually for performance)
Number per Table	Exactly one	Zero, one, or many (depending on relationships)

- **Primary keys** define entity identity and enforce uniqueness.
- **Foreign keys** enforce valid relationships and referential integrity.

In-Depth Study Material: Other Keys & Views

1. Other Keys

Beyond the primary and foreign keys, relational databases recognize several other “key” types that help identify and constrain data.

1.1 Candidate Key

- **Definition** Any column—or minimal set of columns—that **can** uniquely identify each row in a table.
- **Characteristics**
 - Must satisfy **uniqueness** and **non-nullability**.
 - There may be multiple candidate keys in a table.
- **Example**

```
-- In an employees table, both (email) and (social_security_number)
-- could serve as candidate keys.
```

1.2 Alternate Key

- **Definition** A candidate key that is **not** chosen as the table's primary key.
- **Purpose**
 - Enforce uniqueness on columns that aren't the primary key.
 - Provide additional lookup paths.
- **Example**

```
ALTER TABLE employees
ADD CONSTRAINT ak_emp_email UNIQUE (email);
```

1.3 Composite Key

- **Definition** A key composed of **two or more columns** that together uniquely identify a row.
- **Usage**
 - Often used in join tables or to model many-to-many relationships.
- **Example**

```
CREATE TABLE student_courses (
    student_id    INT,
    course_id     INT,
    enrollment_date DATE,
    PRIMARY KEY (student_id, course_id)
);
```

1.4 Unique Key

- **Definition** A constraint that ensures **all values** in the specified column (or set of columns) are **distinct**.
- **Differences from Primary Key**

- Can be **NULL** (depending on the RDBMS).
- A table can have **multiple** unique keys.
- Doesn't automatically enforce non-null unless declared `NOT NULL`.

- **Example**

```
CREATE TABLE products (
  product_id      SERIAL PRIMARY KEY,
  sku             VARCHAR(50) UNIQUE,      -- unique key on SKU
  product_name    VARCHAR(100)
);
```

2. View

A **view** is a **virtual** table representing the result of a stored query. It does not hold data itself but provides a convenient, reusable interface to complex queries.

2.1 Definition

- A named SQL query that appears as a table.
- Data access through a view is translated into the underlying base-table queries at runtime.

2.2 Use Cases

1. **Simplify Complex Queries** Encapsulate lengthy joins, aggregations, or filters for reuse.
2. **Security & Abstraction**
 - Restrict access to certain columns or rows.
 - Present a simplified schema to end users.
3. **Logical Data Independence** Underlying table structures can change while maintaining a stable interface.

2.3 Types of Views

Type	Definition
Simple View	Based on a single table; no grouping or aggregates; may be updatable.
Complex View	Involves multiple tables, joins, aggregates, or set operations; typically read-only.

```
-- Simple view example
CREATE VIEW v_active_customers AS
SELECT id, name, email
FROM customers
WHERE status = 'active';

-- Complex view example
CREATE VIEW v_monthly_sales AS
```



```
SELECT p.product_name,  
       SUM(o.quantity * o.unit_price) AS total_revenue  
FROM products p  
JOIN order_items o ON p.product_id = o.product_id  
GROUP BY p.product_name;
```

2.4 Updatable Views

- **Conditions for Updatability** (may vary by RDBMS):
 - References a **single** base table.
 - Does **not** include `GROUP BY`, `DISTINCT`, aggregates, or `UNION`.
 - Does **not** omit non-nullable columns without defaults.
- **Supported Operations:** `INSERT`, `UPDATE`, `DELETE` on the view propagate to the base table.

2.5 WITH CHECK OPTION

- **Purpose:** Prevents DML through the view that would produce rows **not** satisfying the view's defining `WHERE` clause.
- **Syntax:**

```
CREATE VIEW v_active_products AS  
SELECT * FROM products  
WHERE discontinued = FALSE  
WITH CHECK OPTION;
```

- **Behavior:**
 - **INSERT/UPDATE** through `v_active_products` must have `discontinued = FALSE`, or the operation is rejected.

3. Best Practices & Considerations

- **Key Design**
 - Identify all candidate keys; choose the simplest, most stable attribute(s) as primary key.
 - Index alternate and unique keys to speed lookups.
 - **View Management**
 - Use views to centralize complex logic and enforce consistent filtering.
 - Monitor performance: sometimes a view can hide expensive operations—use `EXPLAIN` to check.
 - Document views thoroughly, indicating purpose and any updatability restrictions.
-

1. Differences between Primary Key, Foreign Key, and Unique Key:

Feature	Primary Key	Foreign Key	Unique Key
Definition	Uniquely identifies each record in a table	Establishes a link between two tables	Ensures all values in a column are unique
Uniqueness	Must be unique and not null	Can contain duplicate values in the referencing table	Must be unique, can be null (once or multiple times depending on DBMS)
Nullability	Not allowed	Allowed (depends on database design)	Allowed
Main Use	Identifies records	Maintains referential integrity	Prevents duplicate values
Example	<code>user_id</code> in <code>Users</code> table	<code>user_id</code> in <code>Orders</code> table referencing <code>Users (user_id)</code>	<code>email</code> column to ensure no duplicate emails

2. How Indexes Affect SELECT and INSERT Queries Differently:

- **SELECT Queries:** Indexes **speed up** data retrieval. When you use a `SELECT` query with a `WHERE`, `ORDER BY`, or `JOIN` clause on indexed columns, the database can quickly find rows without scanning the entire table.
- **INSERT Queries:** Indexes **slow down** `INSERT` operations slightly because the database has to **update the index** each time a new record is added. The more indexes a table has, the more overhead on inserts.

Summary:

- `SELECT` → Faster with indexes
- `INSERT` → Slightly slower due to index maintenance

3. Situation for Using a Composite Key:

A **composite key** is a primary key made of two or more columns.

Example Situation: In a table `StudentCourseRegistration`, you could use `(student_id, course_id)` as a composite key because:

- A student can register for multiple courses.
- A course can have multiple students.
- But **a student can register for the same course only once.**

```
PRIMARY KEY (student_id, course_id)
```

This prevents duplicate course registrations by the same student.

4. Purpose of Using a View and When to Use WITH CHECK OPTION:

- **Purpose of a View:**
 - A **view** is a virtual table based on the result of a `SELECT` query.
 - Useful to:
 - Simplify complex queries.
 - Provide specific access to part of the data (security).
 - Hide underlying table structure.
- **WITH CHECK OPTION:**
 - Ensures that **inserted or updated rows in the view** must **satisfy the view's condition**.
 - Prevents users from inserting data that doesn't match the view's `WHERE` clause.

Example:

```
CREATE VIEW active_users AS
SELECT * FROM users WHERE status = 'active'
WITH CHECK OPTION;
```

This prevents inserting or updating a record through the `active_users` view that would result in a `status` other than `'active'`.

Here are the SQL queries to create the two tables based on your specifications:

1. Create the students Table:

```
CREATE TABLE students (
    student_id INT PRIMARY KEY,
    name VARCHAR(100),
    email VARCHAR(100) UNIQUE,
    status VARCHAR(20)
);
```

- `student_id` is set as the **primary key**.
 - `email` is set as **unique**.
 - `name` and `status` are optional extra columns, assuming you might want them for views or filters.
-

2. Create the enrollments Table with student_id as a Foreign Key:

```
CREATE TABLE enrollments (  
  enrollment_id INT PRIMARY KEY,  
  student_id INT,  
  course_name VARCHAR(100),  
  enrollment_date DATE,  
  FOREIGN KEY (student_id) REFERENCES students(student_id)  
);
```

- student_id is a **foreign key** referencing the students table.
- enrollment_id is a **primary key** for this table.
- course_name and enrollment_date are sample additional columns for a real-world scenario.

Here are the SQL queries for your tasks:

4.1. Create a View active_students Showing Only Students with Active Status:

```
CREATE VIEW active_students AS  
SELECT *  
FROM students  
WHERE status = 'active';
```

Optional: If you want to **prevent insert/update** through this view that changes status to non-active, use WITH CHECK OPTION:

```
CREATE VIEW active_students AS  
SELECT *  
FROM students  
WHERE status = 'active'  
WITH CHECK OPTION;
```

4.2. Create an Index on the email Column of the students Table:

```
CREATE INDEX idx_students_email  
ON students(email);
```

This index helps speed up queries like:

```
SELECT * FROM students WHERE email = 'student@example.com';
```

4.3. Retrieve Student Names from the active_students View:

```
SELECT name  
FROM active_students;
```

This assumes there is a `name` column in the `students` table.

TOPIC 20

1. Coding Standards & Conventions Adhering to a uniform style makes code easier to read, understand, and maintain—especially in teams. Below is a deep dive into the most common conventions in Java (though many principles translate to other languages).

a) Naming Conventions

Entity	Convention	Examples
Classes/Interfaces	PascalCase (capitalize every word)	EmployeeService, OrderDao
Methods/Variables	camelCase (first word lowercase)	calculateSalary(), orderId
Constants	UPPER_SNAKE_CASE (all caps, underscores)	MAX_RETRY_COUNT, PI
Packages	all lowercase, reverse-domain style	com.company.project.module

Why?

- **PascalCase** on types immediately signals “this is a class or interface.”
- **camelCase** on methods/fields differentiates them from types.
- **UPPER_SNAKE_CASE** grabs attention for immutable, shared values.
- **Reverse-domain** ensures uniqueness when publishing libraries.

b) Formatting

- **Indentation:** 4 spaces per level; never mix tabs and spaces.
- **Braces:** Always use braces—even for single statements—to avoid bugs when adding lines later.

```
// ❌ Bad: no braces
if (user != null)
    process(user);

// ✅ Good: always braces
if (user != null) {
    process(user);
}
```

- **Spacing:**
 - Space after keywords: `if (condition), for (i = 0; i < n; i++)`
 - Space around operators: `a + b, count == 0`
 - No trailing spaces at line ends.

- **Line Length:** Aim for ≤ 100 characters; break long expressions into multiple lines at logical points (after commas, before operators).
-

c) Class Structure & File Organization

Within a `.java` file, follow this order:

1. **Package** declaration
2. **Imports** (grouped: standard Java, third-party, project-specific)
3. **Class/interface Javadoc**
4. **Class declaration**
5. **Static fields** (constants first, then other `static`)
6. **Instance fields**
7. **Constructors**
8. **Public methods**
9. **Protected methods**
10. **Private methods**
11. **Inner classes/interfaces**

```
package com.company.project;

import java.util.List;

/**
 * Service for managing employee payroll.
 */
public class PayrollService {
    // 1. Constants
    private static final int MAX_RETRIES = 3;

    // 2. Static variables
    private static DatabaseConnectionPool pool;

    // 3. Instance variables
    private final EmployeeDao employeeDao;

    // 4. Constructors
    public PayrollService(EmployeeDao dao) {
        this.employeeDao = dao;
    }

    // 5. Public methods
    public double calculateMonthlyPay(int employeeId) { ... }

    // 6. Protected methods
    protected void logCalculation(int id, double amount) { ... }

    // 7. Private methods
    private double applyTaxRules(double grossPay) { ... }
}
```

d) Comments & Documentation

- **JavaDoc** for all public APIs, classes, and complex methods.

```
/**
 * Calculates tax based on income and current tax brackets.
 *
 * @param income gross annual income in USD
 * @return tax amount owed
 */
public double calculateTax(double income) { ... }
```

- **Inline comments** only when code is non-obvious. Prefer expressive naming over comments.
-

e) Common Pitfalls & Best Practices

- **Avoid “magic numbers”**—use named constants.
 - **Don’t over-comment** trivial code; instead, refactor to clarify.
 - **Consistent use of logging** rather than `System.out.println`.
 - **Static analysis tools** (Checkstyle, PMD, SpotBugs) to enforce style.
-

2. Unit Testing Guidelines Well-written unit tests act as living documentation, catch regressions early, and give you confidence to refactor.

a) Tooling

- **JUnit 5**: the de facto standard for writing and running tests.
 - **Mockito** (or **EasyMock**) for mocking external dependencies.
 - **AssertJ** or **Hamcrest** for clear, fluent assertions.
-

b) Test Structure & AAA Pattern

Each test should follow **Arrange – Act – Assert**:

1. **Arrange**: set up objects and state.
2. **Act**: perform the action under test.
3. **Assert**: verify the outcome.

```
@Test
void shouldReturnCorrectTaxForStandardIncome() {
    // Arrange
    TaxCalculator calc = new TaxCalculator();
    double income = 50_000;

    // Act
    double tax = calc.calculateTax(income);

    // Assert
    assertThat(tax).isEqualTo(8_250); // e.g., 16.5%
}
```

c) Test Characteristics

- **Isolated**: one unit per test; mock any external system (databases, web services).
 - **Repeatable**: no dependence on environment, file I/O, or time. Use test doubles or clocks for time-sensitive code.
 - **Fast**: tests should run in milliseconds; avoid heavy integration work in unit tests.
 - **Descriptive**: method names that read like sentences, e.g.
`shouldThrowWhenUserNotFound()`.
-

d) Coverage Focus

Aim for meaningful coverage, not 100% blindly:

- **Business logic** and rules.
 - **Boundary conditions** (e.g., zero, max values, empty collections).
 - **Error paths** (exceptions, invalid inputs).
 - **Avoid** testing trivial getters/setters unless they contain logic.
-

e) Test Quality Checklist

- ✓ **Readable**: easy for other developers to understand intent.
 - ✓ **Maintainable**: minimal duplication; use helper methods or fixtures wisely.
 - ✓ **Robust**: fails only when there's a real problem.
 - ✓ **Fast**: keeps the feedback loop tight.
 - ✓ **Independent**: order of execution shouldn't matter.
-

Further Reading & Tools

- **Google Java Style Guide**: <https://google.github.io/styleguide/javaguide.html>
 - **Effective Java** (Joshua Bloch) – best practices in Java.
 - **JUnit 5 User Guide**: <https://junit.org/junit5/docs/current/user-guide/>
 - **Mockito Documentation**: <https://site.mockito.org/>
 - **Checkstyle** configuration for enforcing conventions automatically.
-

3. General Java Best Practices

Good Java code is not just about making it work—it's about keeping it maintainable, robust, and flexible. Below is an in-depth look at key best practices.

a) Code Quality

- **Short, Focused Methods**
 - Aim for **≤20 lines** per method.
 - Each method should do **one thing** (Single Responsibility).
 - Example before/after:

```
// ❌ Too many responsibilities
public void processOrder(Order order) {
    validate(order);
    calculateTotals(order);
    save(order);
    notifyCustomer(order);
}

// ✅ Split into focused methods
public void processOrder(Order order) {
    validate(order);
    calculateTotals(order);
    persist(order);
    notifyCustomer(order);
}
```

- **Prefer Composition over Inheritance**

- Inheritance creates tight coupling; composition (has-a) is more flexible.

```
// ❌ Inheritance for code reuse
class AuditorList extends ArrayList<Record> { ... }

// ✅ Composition instead
class AuditorList {
    private final List<Record> records = new ArrayList<>();
    // delegate methods as needed
}
```

- **Minimize static**

- Use `static` only for true constants or pure utility functions.
- Overusing `static` hinders testability and polymorphism.

- **Avoid null When Possible**

- Return `Optional<T>` instead of `null` to express “value may be absent.”

```
public Optional<User> findUserId(int id) { ... }
```

- **Externalize Hard-Coded Values**

- Move literals into `application.properties`, `YAML`, or private `static final` constants.

```
# application.properties
app.maxLoginAttempts=5
```

- **Use Configuration Classes**

- For complex defaults, leverage Spring’s `@ConfigurationProperties`.

b) Object-Oriented Principles

- **SOLID Principles**

- **S**ingle Responsibility
- **O**pen/Closed
- **L**iskov Substitution
- **I**nterface Segregation
- **D**ependency Inversion

- **Encapsulation**

- Keep fields `private`; expose behavior via getters/setters only if needed.

```
public class Account {  
    private BigDecimal balance;  
    public BigDecimal getBalance() { return balance; }  
    public void deposit(BigDecimal amount) { ... }  
}
```

- **Immutable Classes**

- Make value objects immutable: `final class`, `private final` fields, no setters.

```
public final class Money {  
    private final BigDecimal amount;  
    public Money(BigDecimal amount) { this.amount = amount; }  
    public BigDecimal getAmount() { return amount; }  
}
```

- **Program to an Interface**

- Depend on interfaces, not concrete classes.

```
PaymentProcessor processor = new StripePaymentProcessor();  
// later can swap to PayPalPaymentProcessor without code changes
```

c) Error Handling

- **Never Swallow Exceptions**

- Always log or rethrow with context.

```
try {  
    readFile(path);  
} catch (IOException e) {  
    logger.error("Failed to read file {}", path, e);  
    throw new FileProcessingException("Could not read " + path, e);  
}
```

- **Custom Exceptions for Business Logic**

- Create domain-specific exceptions (e.g., `InsufficientFundsException`).

- **Avoid Empty `catch` Blocks**

- An empty catch hides problems; if truly ignorable, at least comment why.
-

d) Logging

- **Use SLF4J + Logback/Log4J2**

```
private static final Logger logger =  
    LoggerFactory.getLogger(MyClass.class);
```

- **Appropriate Log Levels**

- ERROR for failures
- WARN for recoverable issues
- INFO for high-level flow
- DEBUG for detailed diagnostics

- **Avoid `System.out.println()`**

- Logging frameworks allow level-based filtering and output configuration.
-

e) Code Reviews & Version Control

- **Code Reviews**

- Encourage peer review focusing on:
 - Clarity, correctness, test coverage, adherence to standards
 - Use inline comments (“suggest”, “consider”)—avoid personal bias.

- **Git Best Practices**

- **Feature branches** per ticket/issue
 - **Descriptive commits**: “Add tax calculation for senior citizens”
 - **Small, focused commits** to simplify rebasing and rollback
-

4. Project Structure & Design

A well-organized project structure and clear design patterns pay dividends as applications grow.

a) Modular, Layered Architecture

1. Controller Layer

- Handles HTTP/REST endpoints (`@RestController` in Spring).
- Should contain no business logic—only request parsing and response formatting.

2. Service Layer

- Implements business rules (`@Service`).

- Coordinates between repositories and external systems.

3. DAO/Repository Layer

- Data access logic (@Repository or Spring Data JPA interfaces).
- One interface per aggregate/root entity.

4. Model Layer

- Domain or DTO classes representing data structures.

```
src/
└─ main/
    └─ java/com/company/project/
        ├── controller/
        │   └─ EmployeeController.java
        ├── service/
        │   └─ PayrollService.java
        ├── repository/
        │   └─ EmployeeRepository.java
        └─ model/
            └─ Employee.java
```

b) Dependency Injection

- **Constructor Injection Preferred**

```
@Service
public class PayrollService {
    private final EmployeeRepository repo;
    @Autowired
    public PayrollService(EmployeeRepository repo) {
        this.repo = repo;
    }
}
```

- **Avoid** manual `new` for managed beans—let Spring wire dependencies.

c) Configuration

- **Externalize** environment-specific settings in `application.properties` or `application.yml`.

```
spring:
  datasource:
    url: ${DB_URL}
    username: ${DB_USER}
    password: ${DB_PASS}
```

- **Profiles** for separation:

```
# application-dev.properties
logging.level.root=DEBUG
```

```
# application-prod.properties
logging.level.root=INFO
```

- Use **@Value** or `@ConfigurationProperties` to bind properties to POJOs.

d) Additional Design Considerations

- **Exception Handling**
 - Centralize via `@ControllerAdvice` for REST error responses.
- **Validation**
 - Use `javax.validation` annotations (`@NotNull`, `@Size`, etc.) in DTOs and enforce with `@Valid`.
- **Security**
 - Spring Security for authentication and authorization.
- **Documentation**
 - Swagger/OpenAPI for REST API specs (`springdoc-openapi`).

Here's a comprehensive **in-depth study plan** and resources for mastering **Continuous Learning & Practice in Java**:

5. Continuous Learning & Practice (In-Depth Guide)

To stay relevant and grow as a Java developer, make learning an ongoing habit. This section breaks down key strategies with tools, schedules, and curated resources.

1. Read Java Documentation & Blogs

Why:

- Understand language features deeply
- Stay up-to-date with new versions and industry trends
- Learn how real-world systems are built

What to Follow:

Source	Highlights
Official Java Docs	Authoritative reference for all core Java APIs

Source	Highlights
Baeldung	Practical tutorials on Spring, Java, REST, etc.
InfoQ	Articles on enterprise Java, microservices, architecture
Java Code Geeks	Advanced topics, performance tuning, and tooling
The Java Specialists' Newsletter	Deep dives into language internals
Medium	Community blog posts—diverse topics and styles

2. Practice on Coding Platforms

Why:

- Improve algorithmic thinking and coding speed
- Prepare for job interviews
- Reinforce core Java constructs (collections, OOP, exception handling, etc.)

Where to Practice:

Platform	Best For	Suggested Topics
LeetCode	Algorithms, Data Structures, Interview prep	Arrays, Strings, DP, Trees, Sliding Window
HackerRank	Java domain-specific challenges	OOP, Exceptions, Generics, Regex
Codeforces	Competitive programming	Math-heavy problems, fast input/output
Exercism Java	Test-driven learning, mentoring	Clean coding and design

Practice Routine (Suggested):

- **Daily:** 1-2 problems from LeetCode/HackerRank (easy/medium)
- **Weekly:** 1 mock test or 2-hour challenge session
- **Monthly:** Solve a full project problem (e.g., URL shortener, REST API)

3. Contribute to Open Source

Why:

- Gain real-world experience working in teams
- Learn from code reviews and advanced patterns
- Build a public portfolio

How to Start:

1. **Find projects on GitHub** Search: `language:Java good-first-issues label:help-wanted`

2. **Contribute gradually:**

- Start with fixing typos or docs
- Then try unit tests, refactoring, or bug fixes
- Eventually, contribute features or optimizations

Recommended Java Open Source Projects:

- [Spring Projects](#)
 - [Apache Commons](#)
 - [JHipster](#)
 - [Elasticsearch Java Client](#)
-

4. **Learn One New Tool/Library Every Month**

Why:

- Stay updated with modern ecosystems
- Make your projects cleaner, faster, and more maintainable

Monthly Learning Plan (Example):

Month	Tool/Library	Description
Jan	Lombok	Reduces boilerplate code using annotations
Feb	MapStruct	Type-safe object mapping between DTOs and Models
Mar	Spring Boot	Framework for rapid Java backend development
Apr	Spring Data JPA	Simplifies database interactions
May	Mockito	Mocking framework for unit tests
Jun	Docker	Containerize Java apps for consistent deployment
Jul	Jib	Build Docker images for Java directly from Maven/Gradle
Aug	Flyway/Liquibase	Version control for DB schemas
Sep	Apache Kafka	Event streaming platform
Oct	OpenAPI/Swagger	API documentation
Nov	Hibernate	ORM framework for database persistence
Dec	Keycloak	Open-source Identity and Access Management

Bonus Tips

- **Maintain a Developer Diary:** Note what you learn daily or weekly.
 - **Build Portfolio Projects:** e.g., Bookstore app, Blog engine, Chat app.
 - **Attend Meetups/Webinars:** Join Java User Groups (JUGs), follow conferences like Devoxx, JavaOne.
 - **Read Books:**
 - *Effective Java* by Joshua Bloch
 - *Clean Code* by Robert C. Martin
 - *Java Concurrency in Practice* by Brian Goetz
-