**🧩 1. Single Responsibility Principle (SRP)**

➡ **Meaning:**  
A class should have **only one reason to change**, meaning it should **do only one main job**.

**❌ Bad Example:**

class Report {

void printReport() {

System.out.println("Printing report...");

}

void saveToDatabase() {

System.out.println("Saving report to database...");

}

}

👉 Here, the class is doing **two jobs** — printing and saving.  
If saving logic changes, you must also change this class — that’s bad.

**✅ Good Example:**

class ReportPrinter {

void printReport() {

System.out.println("Printing report...");

}

}

class ReportSaver {

void saveToDatabase() {

System.out.println("Saving report to database...");

}

}

Now, each class does **only one job** — easier to manage and test.

**🧩 2. Open/Closed Principle (OCP)**

➡ **Meaning:**  
Your code should be **open for adding new features**, but **closed for changing existing code**.

**❌ Bad Example:**

class AreaCalculator {

double calculate(Object shape) {

if (shape instanceof Circle) {

Circle c = (Circle) shape;

return Math.PI \* c.radius \* c.radius;

} else if (shape instanceof Rectangle) {

Rectangle r = (Rectangle) shape;

return r.length \* r.width;

}

return 0;

}

}

👉 If you add a new shape, you must **change this class** — not good.

**✅ Good Example:**

interface Shape {

double area();

}

class Circle implements Shape {

double radius;

Circle(double r) { radius = r; }

public double area() {

return Math.PI \* radius \* radius;

}

}

class Rectangle implements Shape {

double length, width;

Rectangle(double l, double w) { length = l; width = w; }

public double area() {

return length \* width;

}

}

class AreaCalculator {

double calculate(Shape shape) {

return shape.area(); // No need to change for new shapes

}

}

Now, if you add a new shape, just **create a new class** — no need to edit the calculator.

**🧩 3. Liskov Substitution Principle (LSP)**

➡ **Meaning:**  
A subclass should be able to **replace its parent class** without breaking the program.

**❌ Bad Example:**

class Bird {

void fly() {

System.out.println("Flying...");

}

}

class Penguin extends Bird {

void fly() { // Penguins can't fly!

throw new UnsupportedOperationException("Penguins can't fly!");

}

}

👉 This breaks the rule — because you **can’t replace Bird with Penguin** safely.

**✅ Good Example:**

interface Bird {

void eat();

}

interface FlyingBird extends Bird {

void fly();

}

class Sparrow implements FlyingBird {

public void eat() { System.out.println("Sparrow eating"); }

public void fly() { System.out.println("Sparrow flying"); }

}

class Penguin implements Bird {

public void eat() { System.out.println("Penguin eating"); }

}

Now, we separated flying birds and non-flying birds — no issue!

**🧩 4. Interface Segregation Principle (ISP)**

➡ **Meaning:**  
Don’t make one big interface with too many methods.  
Instead, make **small, specific interfaces** so classes don’t have to implement things they don’t need.

**❌ Bad Example:**

interface Worker {

void work();

void eat();

}

class Robot implements Worker {

public void work() { System.out.println("Robot working"); }

public void eat() { /\* Robots don’t eat \*/ }

}

**✅ Good Example:**

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 what it really needs.

**🧩 5. Dependency Inversion Principle (DIP)**

➡ **Meaning:**  
High-level classes should **not depend on low-level classes**.  
Both should depend on **abstractions (interfaces)**.

**❌ Bad Example:**

class Keyboard { }

class Monitor { }

class Computer {

private Keyboard keyboard;

private Monitor monitor;

Computer() {

keyboard = new Keyboard(); // directly depending

monitor = new Monitor();

}

}

👉 Hard to change Keyboard or Monitor later.

**✅ Good Example:**

interface Keyboard { }

interface Monitor { }

class WiredKeyboard implements Keyboard { }

class HDMonitor implements Monitor { }

class Computer {

private Keyboard keyboard;

private Monitor monitor;

Computer(Keyboard keyboard, Monitor monitor) {

this.keyboard = keyboard;

this.monitor = monitor;

}

}

Now, you can easily **swap** the keyboard or monitor type — flexible and testable.

**🎯 Summary Table**

| **Principle** | **Goal** | **Simple Tip** |
| --- | --- | --- |
| **S** | Single Responsibility | One class = One job |
| **O** | Open/Closed | Add new, don’t edit old |
| **L** | Liskov Substitution | Subclasses should behave like parents |
| **I** | Interface Segregation | Keep interfaces small |
| **D** | Dependency Inversion | Depend on interfaces, not classes |

Would you like me to make a **diagram or flowchart** to visualize all SOLID principles easily? It can help you remember them better.

## 💡 What is Encapsulation in Java?

**Encapsulation** in Java means **binding data (variables)** and **methods (functions)** that operate on that data into a **single unit (class)**.

It also means **hiding the data** from outside access and **allowing controlled access** through methods.

🧩 In short:  
**Encapsulation = Data hiding + Controlled access using getter and setter methods**

## 🧱 Real-life Example

Think of a **bank account**:

* You can **deposit** and **withdraw** money using methods.
* But you **can’t directly change the balance** from outside.

That’s what encapsulation does!

**🧩 2. Inheritance — *"Reusing the Code"***

**💡 Meaning:**

Inheritance allows one class (child/subclass) to **inherit** the fields and methods of another class (parent/superclass).

In simple words:

Child class can use parent’s properties and behaviors.

**🧱 Example:**

// Parent class

class Animal {

void eat() {

System.out.println("This animal eats food");

}

}

// Child class

class Dog extends Animal {

void bark() {

System.out.println("Dog barks");

}

}

public class Main {

public static void main(String[] args) {

Dog d = new Dog();

d.eat(); // inherited from Animal

d.bark(); // Dog's own method

}

}

**✅ Why it’s useful:**

* **Reusability:** You don’t need to write the same code again.
* **Maintainability:** Common behavior can stay in the parent.
* **Extensibility:** You can easily add new features using subclasses.

**🧩 3. Polymorphism — *"One thing, many forms"***

**💡 Meaning:**

Polymorphism means **same function name**, but **different behavior** depending on the object.

In simple words:

Many forms of the same thing.

**📍 Types of Polymorphism:**

| **Type** | **When it Happens** | **Example** | **Also Called** |
| --- | --- | --- | --- |
| **Compile-time Polymorphism** | During compilation | Method Overloading | Static Polymorphism |
| **Runtime Polymorphism** | During program execution | Method Overriding | Dynamic Polymorphism |

**🧱 Example 1 – Method Overloading (Compile-time)**

Same method name, but different parameters.

class MathOperation {

int add(int a, int b) {

return a + b;

}

int add(int a, int b, int c) {

return a + b + c;

}

}

public class Main {

public static void main(String[] args) {

MathOperation m = new MathOperation();

System.out.println(m.add(2, 3)); // calls 2-arg method

System.out.println(m.add(2, 3, 4)); // calls 3-arg method

}

}

**🧱 Example 2 – Method Overriding (Runtime)**

## 💡 What is ****Method Overriding****?

**Method overriding** happens when a **subclass (child class)** provides a **new (its own) implementation** of a method that is **already defined in the superclass (parent class)**.

In other words:

The **same method name** and **same parameters**, but **different behavior** in the child class.

class Animal {

void sound() {

System.out.println("Animal makes sound");

}

}

class Dog extends Animal {

void sound() {

System.out.println("Dog barks");

}

}

public class Main {

public static void main(String[] args) {

Animal a = new Dog(); // upcasting

a.sound(); // Dog’s version runs (runtime polymorphism)

}

}

**✅ Why it’s useful:**

* Increases **flexibility**.
* Makes code **extensible** and **easier to maintain**.
* Supports **dynamic behavior** (based on object type).

# 🧠 1. **When to Use Abstract Class**

### ✅ Use an abstract class when:

1. You want **partial abstraction** — i.e., some methods have implementation, some don’t.
2. You want to **share code(properties and behaviour) among related classes**.
3. You expect **subclasses to inherit common behavior**.
4. You have **state/fields (variables)** that should be shared among subclasses.

### 📘 Example of Abstract Class

// Abstract class

abstract class Vehicle {

String color; // shared property

// Constructor

Vehicle(String color) {

this.color = color;

}

// Abstract method (must be implemented by subclasses)

abstract void start();

// Normal method (shared by all)

void stop() {

System.out.println("Vehicle stopped");

}

}

// Subclass

class Car extends Vehicle {

Car(String color) {

super(color);

}

void start() {

System.out.println(color + " car started with key");

}

}

class Bike extends Vehicle {

Bike(String color) {

super(color);

}

void start() {

System.out.println(color + " bike started with button");

}

}

public class Main {

public static void main(String[] args) {

Vehicle car = new Car("Red");

car.start();

car.stop();

Vehicle bike = new Bike("Blue");

bike.start();

bike.stop();

}

}

**Why abstract class here?**

* Vehicle has a **shared property** (color) and **shared behavior** (stop()).
* Each subclass implements its **own start() method**.
* Partial abstraction makes sense here.

# 🧠 2. **When to Use Interface**

### ✅ Use an interface when:

1. You want **full abstraction** — i.e., just define rules, no implementation (except default/static methods).
2. You want to implement **multiple inheritance** (a class can implement multiple interfaces).
3. You want to **decouple code** — e.g., define a contract without worrying about implementation.
4. You don’t need **state/fields**, only behavior.

### 📘 Example of Interface

// Interface

interface Payment {

void pay(double amount); // abstract method

}

// Implementing classes

class CreditCardPayment implements Payment {

public void pay(double amount) {

System.out.println("Paid " + amount + " using Credit Card");

}

}

class UpiPayment implements Payment {

public void pay(double amount) {

System.out.println("Paid " + amount + " using UPI");

}

}

public class Main {

public static void main(String[] args) {

Payment p1 = new CreditCardPayment();

p1.pay(500);

Payment p2 = new UpiPayment();

p2.pay(200);

}

}

**Why interface here?**

* Payment types (CreditCard, UPI, Wallet, etc.) **share a common behavior**: pay().
* Each payment method implements pay() **its own way**.
* Multiple unrelated classes can implement the same interface.

# 🧠 1. Java ka Standard Main Method

Java program **execution start hoti hai** main method se. Standard declaration hoti hai:

public static void main(String[] args) {

// program starts here

}

**Explanation of keywords:**

| **Keyword** | **Meaning** |
| --- | --- |
| public | JVM ko access allow karta hai. |
| static | JVM bina object banaye directly call kar sakta hai. |
| void | Ye method return kuch nahi karta. |
| main | JVM specially recognize karta hai. |
| String[] args | Command-line arguments receive karne ke liye. |

# 🧠 2. Agar Dusra Main Method Ho

### Case 1: ****Same class me overloaded main****

public class Test {

public static void main(String[] args) {

System.out.println("Original main");

main(10); // call overloaded main

}

// Overloaded main method

public static void main(int x) {

System.out.println("Overloaded main: " + x);

}

}

**Output:**

Original main

Overloaded main: 10

**Explanation:**

* JVM **sirf public static void main(String[] args)** ko execution start ke liye call karta hai.
* Aap **apna overloaded main** create kar sakte ho (main(int x)), lekin JVM automatically call nahi karega.
* Aap chahe to normal method ki tarah manually call kar sakte ho.

# 🧩 **1. JVM (Java Virtual Machine)**

### 🔹 Definition:

JVM ek **virtual machine** hai jo **Java bytecode ko execute** karti hai.

### 🔹 Features:

* Platform independent → "Write once, run anywhere".
* **Bytecode ko machine code me translate** karke execute karta hai.
* Responsible for **memory management**, **garbage collection**, etc.

### 🔹 Analogy:

JVM ko aise samjho jaise **engine** jo car chalati hai. Engine directly **fuel ko motion me convert karta hai**, JVM **bytecode ko run karta hai**.

# 🧩 **2. JRE (Java Runtime Environment)**

### 🔹 Definition:

JRE ek **environment hai jo Java programs ko run karne ke liye** chahiye.

### 🔹 Features:

* JVM included hoti hai.
* Java program ke execution ke liye **libraries, class files, Java API** provide karta hai.
* **Run time ke liye sufficient** hai, but program **compile nahi kar sakte**.

### 🔹 Analogy:

JRE ko samjho jaise **car ka fuel + engine + dashboard**.

* Engine → JVM
* Fuel & dashboard → libraries & API jo car chalane me help karti hai.

# 🧩 **3. JDK (Java Development Kit)**

### 🔹 Definition:

JDK ek **complete toolkit** hai jo **Java program ko develop aur run** karne ke liye chahiye.

### 🔹 Features:

* JDK = **JRE + Development tools**
* Tools include:
  + javac → compiler (source code → bytecode)
  + javadoc → documentation generator
  + jar → archive tool
  + And other development tools.
* Basically **code likhne + compile + run** sab kuch possible.

### 🔹 Analogy:

JDK ko samjho jaise **garage + engine + fuel + mechanic tools**.

* Garage → development environment
* Tools → compiler, debugger etc.
* Engine → JVM
* Fuel → JRE libraries

# 🔗 **Flow: Java Program Execution**

Chalo ab step by step **flow diagram ki tarah** samjhte hain:

1. **Program likhte ho** → HelloWorld.java (source code)
2. **Compile karte ho JDK se** → javac HelloWorld.java
   * Compiler bytecode (.class) generate karta hai.
3. **Run karte ho JVM se** → java HelloWorld
   * JVM bytecode ko read karke machine code me convert karta hai aur execute karta hai.
4. **Execution ke liye JRE** ka environment use hota hai (libraries, JVM included).

# 🧩 **1. Static Method Kya Hai?**

Static method wo method hota hai jo **class ke saath belong karta hai, object ke saath nahi**.

**Key Points:**

* static keyword use hota hai.
* Object create kiye bina directly call kar sakte ho.
* Class level method → sab objects me shared hota hai.
* Static method ke andar **instance variables (non-static)** directly access nahi kar sakte.

# 🧩 **1. Instance Variable Kya Hai?**

Instance variable wo variable hota hai jo **har object ke liye alag hota hai**.

**Key Points:**

* Object ke **state/attribute** ko store karte hain.
* **Class ke andar declare** hota hai, but **method ke andar nahi**.
* **Har object ka apna copy** hota hai.
* Non-static hota hai → object create karna zaruri hai access karne ke liye.

# 🧩 **1. Call by Value (Value ke through)**

Java me **primitive data types** (int, float, boolean etc.) **Call by Value** use karte hain.

**Kya hota hai:**

* Original value **change nahi hoti**.
* Method me **copy of value pass hoti hai**, original variable safe rehta hai.

### 📘 Example:

public class Main {

static void changeValue(int a) {

a = a + 10; // sirf copy change hoti hai

}

public static void main(String[] args) {

int num = 50;

changeValue(num);

System.out.println("Num: " + num); // 50, original change nahi hua

}

}

**Explanation:**

* num ka value 50 hai.
* changeValue method me a ka copy banta hai → 50 → 60.
* Original num **unchanged** rehta hai.

✅ Primitive types → **Call by Value**

# 🧩 **2. Call by Reference (Reference ke through)**

Java me objects **Call by Reference** ki tarah behave karte hain.

**Kya hota hai:**

* Method me **object reference pass hota hai**.
* Original object ka **state change ho sakta hai**.

### 📘 Example:

class Car {

String color;

}

public class Main {

static void changeColor(Car c) {

c.color = "Blue"; // original object change ho raha hai

}

public static void main(String[] args) {

Car car1 = new Car();

car1.color = "Red";

changeColor(car1);

System.out.println("Car color: " + car1.color); // Blue

}

}

**Explanation:**

* car1 ek object hai.
* Method me reference c pass hota hai → original object ka **state change** hota hai.
* Output me color **Red → Blue**.

✅ Objects → **Call by Reference (actually reference pass hota hai)**

# 🧩 **1. ClassNotFoundException**

### 🔹 Definition:

ClassNotFoundException ek **checked exception** hai jo **tab throw hoti hai jab JVM tries to load a class dynamically and it does not found in classpath**

### 🔹 Key Points:

* Checked exception → **compile time pe handle karna padta hai** (try-catch ya throws).
* Mostly **Reflection ya dynamic loading** me hota hai.
* Class file exist nahi karta ya classpath me missing hota hai.

### 🔹 Example:

public class Main {

public static void main(String[] args) {

try {

Class.forName("com.example.MyClass"); // dynamically load

} catch (ClassNotFoundException e) {

System.out.println("Class not found!");

}

}

}

**Explanation:**

* Agar com.example.MyClass class available nahi → JVM **ClassNotFoundException** throw karega.

✅ **Use case:** Reflection, JDBC driver loading etc.

# 🧩 **2. NoClassDefFoundError**

### 🔹 Definition:

NoClassDefFoundError ek **runtime error** hai jo **JVM ko class compile time pe mili thi, lekin runtime pe nahi mili**.

### 🔹 Key Points:

* **Unchecked error** → compile time me nahi pata chalta.
* Mostly **classpath problem** ya **JAR missing** ki wajah se hota hai.
* Class **compile time pe exist karti thi**, runtime pe missing hai.

### 🔹 Example:

// Compile time pe available thi

class MyClass {

void display() {

System.out.println("Hello");

}

}

// Runtime me agar JVM ko MyClass.class nahi mile

public class Main {

public static void main(String[] args) {

MyClass obj = new MyClass(); // NoClassDefFoundError

obj.display();

}

}

**Explanation:**

* JVM ko compile time pe class mili, compile successful → run karte time agar class file missing → **NoClassDefFoundError**

✅ **Common causes:**

* Classpath set nahi hai
* JAR file missing hai
* Deployment me class file delete ho gayi

# 🧩 **1. Comparable Interface**

### 🔹 Definition:

Comparable interface **class ke andar implement** kiya jata hai, aur **natural ordering** define karta hai.

### 🔹 Key Points:

* **Interface** hai → compareTo() method define karna padta hai.
* Object khud decide karta hai kaise compare hoga (natural order).
* **Single sorting sequence** possible.
* **java.lang package** me hai.

### 🔹 Syntax

class Student implements Comparable<Student> {

String name;

int marks;

Student(String name, int marks) {

this.name = name;

this.marks = marks;

}

@Override

public int compareTo(Student s) {

return this.marks - s.marks; // ascending order by marks

}

}

### 🔹 Example Usage

import java.util.\*;

public class Main {

public static void main(String[] args) {

List<Student> students = new ArrayList<>();

students.add(new Student("Amit", 80));

students.add(new Student("Ravi", 95));

students.add(new Student("Sita", 90));

Collections.sort(students); // Comparable use

for(Student s : students) {

System.out.println(s.name + " " + s.marks);

}

}

}

**Output:**

Amit 80

Sita 90

Ravi 95

✅ **Natural sorting** → marks ke ascending order me sort hua.

# 🧩 **2. Comparator Interface**

### 🔹 Definition:

Comparator interface **external sorting** ke liye use hota hai.

### 🔹 Key Points:

* Class ke **bahar** implement kiya jata hai.
* Multiple sorting sequences possible.
* compare() method implement karna padta hai.
* **java.util package** me hai.

### 🔹 Syntax

class SortByName implements Comparator<Student> {

@Override

public int compare(Student s1, Student s2) {

return s1.name.compareTo(s2.name); // sort by name

}

}

# 🧩 **1. equals() Method**

### 🔹 Definition:

equals() method **do objects ko compare karne ke liye** use hota hai.

* Ye **Object class** ka method hai → sabhi classes me available.
* Default behavior: **reference compare karta hai** (==) → same memory location check hoti hai.
* Hum isko **override karke logical comparison** kar sakte hain.

### 🔹 Syntax

public boolean equals(Object obj)

* Return type = boolean → true/false
* Parameter = Object type

### 🔹 Example (Without Override)

class Student {

String name;

int marks;

Student(String name, int marks) {

this.name = name;

this.marks = marks;

}

}

public class Main {

public static void main(String[] args) {

Student s1 = new Student("Amit", 90);

Student s2 = new Student("Amit", 90);

System.out.println(s1.equals(s2)); // false (default == check)

}

}

* Yaha **false** because reference different hai, object same content ke sath bhi memory me alag hai.

### 🔹 Example (With Override)

class Student {

String name;

int marks;

Student(String name, int marks) {

this.name = name;

this.marks = marks;

}

@Override

public boolean equals(Object obj) {

if(this == obj) return true;

if(obj == null || getClass() != obj.getClass()) return false;

Student s = (Student) obj;

return marks == s.marks && name.equals(s.name);

}

}

public class Main {

public static void main(String[] args) {

Student s1 = new Student("Amit", 90);

Student s2 = new Student("Amit", 90);

System.out.println(s1.equals(s2)); // true

}

}

✅ Logical equality → content same → true

# 🧩 **2. hashCode() Method**

### 🔹 Definition:

hashCode() method **object ka integer hash value return karta hai**.

* Ye **Object class ka method** hai.
* JVM isko **hash-based collections** me use karta hai (HashMap, HashSet, Hashtable).

### 🔹 Syntax

public int hashCode()

* Return type = int

### 🔹 Important Rule

Agar equals() override kiya hai → **hashCode() bhi override karna chahiye**

* **Reason:** Hash-based collection (HashSet, HashMap) me **objects ke location decide karne ke liye hashCode use hota hai**.
* Agar equal objects ka hashCode different hoga → collection properly work nahi karega.

### 🔹 Example (With hashCode)

class Student {

String name;

int marks;

Student(String name, int marks) {

this.name = name;

this.marks = marks;

}

@Override

public boolean equals(Object obj) {

if(this == obj) return true;

if(obj == null || getClass() != obj.getClass()) return false;

Student s = (Student) obj;

return marks == s.marks && name.equals(s.name);

}

@Override

public int hashCode() {

return name.hashCode() + marks;

}

}

public class Main {

public static void main(String[] args) {

Student s1 = new Student("Amit", 90);

Student s2 = new Student("Amit", 90);

HashSet<Student> set = new HashSet<>();

set.add(s1);

set.add(s2);

System.out.println(set.size()); // 1

}

}

* Agar hashCode() override nahi hota → size 2 aata
* Override karne ke baad → size 1, kyunki logical equality ke hisaab se same object hai

# 🧩 **3. Summary / Key Points**

| **Method** | **Purpose** | **Default Behavior** | **Override Kab Kare** |
| --- | --- | --- | --- |
| equals() | Compare two objects logically | Reference check (==) | Jab object content comparison chahiye |
| hashCode() | Hash-based collections me location decide | JVM generate karta hai memory address ke basis pe | Agar equals() override ho, to hashCode bhi override kare |

# 🧩 **4. Java me facility**

* **HashMap / HashSet / Hashtable** → fast search, insert, delete
* **equals() + hashCode() override** → ensures **logical equality & consistent hashing**
* **Collections / data structures me object uniqueness maintain**

💡 **Trick yaad rakhne ke liye:**

1. equals() = **"Do objects logically same hain?"**
2. hashCode() = **"Inka unique integer code kya hai?"**
3. **Both override together** → HashSet/HashMap properly kaam kare

Agar chaho mai ek **diagram bana du** jo **equals() aur hashCode() ka role HashSet me visually** dikhaye, taaki easily yaad rahe.

Chahiye mai wo bana du?

##### You said:

🔹 Important Rule Agar equals() override kiya hai → hashCode() bhi override karna chahiye Reason: Hash-based collection (HashSet, HashMap) me objects ke location decide karne ke liye hashCode use hota hai. Agar equal objects ka hashCode different hoga → collection properly work nahi karega. smjha

###### ChatGPT said:

Bilkul! Chalo ise **bohot simple Hinglish me step by step** samjhte hain.

# 🔹 Concept:

1. **Hash-based collections** jaise **HashSet, HashMap** objects ko **hashCode ke basis pe store** karte hain.
   * Matlab: JVM object ko ek **bucket / location** me store karta hai, jiska address **hashCode se decide hota hai**.
2. **equals()** method check karta hai **do objects logically same hain ya nahi**.

# 🔹 Problem:

* Agar hum **equals() override karte hain** (logical equality define ki), lekin **hashCode override nahi karte**, to hash-based collection **galat behave karegi**.

### Example:

class Student {

String name;

int marks;

Student(String name, int marks) {

this.name = name;

this.marks = marks;

}

@Override

public boolean equals(Object obj) {

Student s = (Student) obj;

return this.name.equals(s.name) && this.marks == s.marks;

}

// hashCode() override nahi kiya

}

HashSet<Student> set = new HashSet<>();

Student s1 = new Student("Amit", 90);

Student s2 = new Student("Amit", 90);

set.add(s1);

set.add(s2);

System.out.println(set.size()); // 2 !!! 😱

* Logical equality ke hisaab se s1 aur s2 **same** hain (equals() = true)
* Lekin **hashCode same nahi hai** → JVM ne **different buckets** me store kar diya
* Result → HashSet me duplicate allowed ho gaya

# 🔹 Solution:

@Override

public int hashCode() {

return name.hashCode() + marks;

}

* Ab s1 aur s2 ka **hashCode same** hai → same bucket me check hota hai → duplicates avoid hote hain

System.out.println(set.size()); // 1 ✅

# 🔹 Simple Rule yaad rakhne ke liye:

1. equals() = “Do objects logically same hain?”
2. hashCode() = “Unka bucket / location same hai?”
3. **Equals override → hashCode bhi override** → tab HashSet, HashMap properly kaam kare

Agar chaho, mai ek **diagram bana du** jisme **HashSet me equals + hashCode ka kaam visually** dikhaya ho, taaki easily yaad rahe.

Chahiye mai wo bana du?

##### You said:

internal working btana kaise hota hai hashmap mai acche se

###### ChatGPT said:

Bilkul! Chalo **HashMap ka internal working** step by step samjhte hain **Hinglish me**. Main simple terms + flow ke saath explain karunga.

# 🧩 **HashMap ka internal structure**

1. **HashMap = Array + Linked List / Tree ka combination**
2. Java 8 se agar **bucket me nodes ka size > 8**, to LinkedList **Tree (Red-Black Tree)** me convert ho jata hai → faster search

# 🔹 **1. Key Concept**

* **HashMap me data store hota hai key-value pair ke form me**
* Example:

HashMap<String, Integer> map = new HashMap<>();

map.put("Amit", 90);

map.put("Ravi", 95);

* "Amit" = key, 90 = value

# 🔹 **2. How HashMap Stores Data**

1. **hashCode() calculate hota hai key ke liye**

int hash = key.hashCode();

* Ye integer value decide karta hai **bucket / index** jahan object store hoga

1. **Index calculate hota hai** (array me position ke liye):

int index = hash % array.length;

* Matlab array ke kisi bucket me value store hogi

1. **Bucket me check hota hai**:

* Agar bucket empty hai → **new Node create** karke store kar do
* Agar bucket me already Node hai →
  + equals() check hota hai → key same hai ya nahi
  + Agar same → value **update** kar do
  + Agar different → **linked list / tree me add**

# 🔹 **3. HashMap Retrieval (get())**

1. get(key) call hota hai → key ka **hashCode calculate** hota hai
2. **Bucket / index find hota hai** array me
3. Bucket me Node check hota hai → equals() ke through match hota hai
4. Match mile → **value return**
5. Agar Node nahi mila → **null return**

# 🔹 **4. Collisions**

* Jab do keys ka **hashCode same hota hai** → same bucket me store hote hain
* Solution: **LinkedList** me add karte hain
* Java 8 → agar linked list > 8 nodes → **Red-Black Tree** me convert → O(log n) lookup

# 🔹 **5. Load Factor & Rehashing**

1. **Default initial capacity = 16**, **load factor = 0.75**
2. Jab elements ka number = capacity \* load factor → **rehashing** hota hai
   * Array ka size double hota hai
   * Saare elements **new array me redistribute** hote hain using new index

# 🔹 **6. Summary Flow Diagram (Text Form)**

put(key, value):

1. key.hashCode() -> calculate hash

2. index = hash % array.length

3. bucket check:

- empty -> add node

- not empty -> linked list/tree

- equals() check

- key same -> update value

- key different -> add node

get(key):

1. key.hashCode() -> calculate hash

2. index = hash % array.length

3. bucket check linked list/tree

4. equals() match -> return value

5. else null

# 🔹 **Important Points**

1. hashCode() = **bucket location decide**
2. equals() = **key equality check**
3. **Collision** = same bucket → linked list / tree
4. **Load Factor** = resizing rule → performance optimization

💡 **Trick yaad rakhne ke liye**:

* **HashCode → index**
* **Equals → key match**
* **Collision → linked list / tree**
* **Load factor → rehashing**

Agar chaho, mai **ek neat diagram bana du jo HashMap ka internal working + collision + rehashing + tree conversion ek saath visually dikhaye**, taaki sab easily samajh aaye.

Chahiye mai wo bana du?

##### You said:

equls phle chalta hai ya hashcode phle

###### ChatGPT said:

Bahut achha question hai! Chalo **step by step** samjhte hain ki **HashMap me equals() aur hashCode() ka sequence** kya hai.

# 🔹 **1. put() case (key-value insert)**

1. **hashCode() pehle call hota hai**
   * JVM ye decide karta hai ki **key ko kis bucket / index me store karna hai**
   * Example:

int hash = key.hashCode();

int index = hash % array.length; // bucket index

1. **Bucket check**
   * Agar bucket me **koi existing node hai** → equals() call hota hai
   * equals() decide karta hai **key same hai ya nahi**
2. **Decision**
   * equals() = true → value update
   * equals() = false → naya node add

# 🔹 **2. get() case (value retrieve)**

1. **hashCode() pehle call hota hai**
   * Bucket / index find karne ke liye
2. **Bucket me node(s) check**
   * Agar multiple nodes hai → equals() call hota hai key match ke liye
3. **Decision**
   * equals() = true → value return
   * equals() = false → next node check

# 🔹 **Summary**

| **Operation** | **Step 1** | **Step 2** |
| --- | --- | --- |
| put() | hashCode() → find bucket | equals() → check key equality |
| get() | hashCode() → find bucket | equals() → check key match |

✅ **Conclusion:**

**hashCode() pehle chalta hai**, phir **equals()** tabhi call hota hai jab **same bucket me multiple nodes** ho.

💡 **Trick yaad rakhne ke liye:**

* **hashCode → bucket location**
* **equals → logical key match**

Agar chaho, mai **HashMap me hashCode + equals ka working diagram** bana du, jisse visually easily samajh aaye ki pehle kya hota hai aur phir kya hota hai.

Chahiye mai wo bana du?

##### You said:

ek entry aati hai aur based on the hashcode wo bucket mai add ho jaati hai aur fir dobara entry aati hai lekin iss baar key same hai lekin hashcode differnt aisa possible hai kya ?

###### ChatGPT said:

Achha sawal hai! Chalo **step by step samjhte hain**.

# 🔹 **1. HashCode ka role**

* HashMap me **bucket decide karne ke liye hashCode() use hota hai**.
* Agar key ka hashCode **same nahi** → JVM **alag bucket** me store karega.

# 🔹 **2. HashCode aur equals ki rule**

Java ki **contract** ke hisaab se:

1. Agar equals() kehta hai **two objects equal hain**, to **unke hashCode same hone chahiye**.
2. Agar hashCode different hai → HashMap me same object **duplicate add ho sakta hai**.

Matlab:

a.equals(b) = true --> a.hashCode() == b.hashCode() must be true

* Ye **Java specification ka rule hai**.
* Agar ye rule break hota hai → HashMap **expected behavior nahi dega**.

# 🔹 **3. Realistic scenario**

* Agar equals() override karke **logical equality define ki** hai, lekin **hashCode override nahi kiya** → default Object.hashCode() use hota hai → **different memory ke basis pe hashCode generate hota hai**.
* Is case me **equal objects ke different hashCodes** → HashMap me **alag-alag buckets** me store ho jayenge → **duplicate entry**

class Student {

String name;

@Override

public boolean equals(Object obj) {

Student s = (Student) obj;

return this.name.equals(s.name);

}

// hashCode() override nahi kiya

}

Student s1 = new Student("Amit");

Student s2 = new Student("Amit");

HashMap<Student, Integer> map = new HashMap<>();

map.put(s1, 90);

map.put(s2, 95); // both go to different buckets

System.out.println(map.size()); // 2 ❌ duplicate

✅ **Conclusion:**

* **Technically possible** ki equals() true ho aur hashCode different ho → JVM ke rule break karne par duplicate add ho jayega.
* **Correct Java practice:**
  + Agar equals() override karo → **hashCode() bhi override karo**
  + Dono consistent ho → HashMap / HashSet properly kaam kare

💡 **Rule yaad rakhne ka trick:**

* **Equals decide kare “logic same hai ya nahi”**
* **HashCode decide kare “bucket location”**
* **Contract:** equals true → hashCode same

Agar chaho mai ek **diagram bana du jisme HashMap me duplicate add hone ka scenario dikhaya ho jab hashCode inconsistent ho**, taaki easily samajh aaye.

Chahiye mai wo bana du?

##### You said:

means hashcode same to index same isliye equals se chck hoga ki entry same hai ki different

###### ChatGPT said:

Bilkul! 👌 Ye bilkul sahi samjha aapne. Chalo ise **step by step simple flow** me confirm kar lete hain:

# 🔹 **HashMap me entry insert ka flow**

1. **Key ka hashCode calculate hota hai** → ye decide karta hai **bucket / index**
   * **hashCode same → same bucket**
   * **hashCode different → different bucket**
2. **Bucket me check hota hai**
   * Agar **bucket me pehle se koi Node hai** → equals() call hota hai
     + equals() = true → existing value update
     + equals() = false → new node add
3. Agar **bucket empty hai** → new node direct add

# 🔹 **Important Points**

* **hashCode → bucket location decide karta hai**
* **equals → bucket me objects ko logically compare karta hai**
* HashMap ka fayda → **fast access O(1) average**, kyunki equals() sirf **same bucket me objects** pe check hota hai

💡 **Trick yaad rakhne ke liye:**

hashCode → "Kaunsa bucket"

equals → "Same bucket me key same hai ya nahi"

Agar chaho, mai ek **visual diagram bana du jo hashCode + equals + bucket flow dikhaye**, jisse HashMap ka internal working ek nazar me samajh aaye.

Chahiye mai wo bana du?

##### You said:

collision k case mai kya hota hai

###### ChatGPT said:

Bilkul! Chalo **HashMap me collision** ko **bohot simple aur step-by-step** samjhte hain.

# 🔹 **1. Collision ka matlab**

Collision tab hota hai jab **do different keys ka hashCode same ho** → same **bucket / index** pe store hone ki situation.

# 🔹 **2. Collision ka handling HashMap me**

### Java 8+ ke liye:

1. **Bucket me Node check hota hai**
2. **Bucket empty →** nayi Node add
3. **Bucket me already Node(s) →** linked list me add
4. **Extra:**
   * Agar **linked list ka size > 8**, to LinkedList → **Red-Black Tree** me convert
   * Lookup faster ho jata hai → O(log n)

# 🔹 **3. Example of Collision**

class Key {

int id;

Key(int id) { this.id = id; }

@Override

public int hashCode() {

return 1; // deliberately same hashCode for all keys

}

@Override

public boolean equals(Object obj) {

Key k = (Key)obj;

return this.id == k.id;

}

}

HashMap<Key, String> map = new HashMap<>();

map.put(new Key(1), "A");

map.put(new Key(2), "B"); // same bucket because hashCode = 1

map.put(new Key(3), "C"); // same bucket

* **Sab keys ka hashCode same hai → same bucket**
* JVM **linked list** me add karta hai
* Jab get() call hota hai → linked list me equals() se key check hoti hai → correct value return hoti hai

# 🔹 **4. Collision ka internal flow**

put(key, value):

1. key.hashCode() -> calculate bucket

2. bucket check

- empty -> add node

- not empty -> traverse linked list

- equals() check

- key match -> update value

- key not match -> add new node at end

3. If list size > 8 -> convert to tree

get(key):

1. key.hashCode() -> bucket

2. traverse list/tree

3. equals() check -> return value

# 🔹 **5. Important Notes**

* Collision **normal hai** → hash function pe depend karta hai
* Proper **hashCode implementation** → collision kam kar deta hai → better performance
* **Java 8+ tree conversion** → worst-case O(log n) lookup

💡 **Trick yaad rakhne ke liye:**

* **hashCode → bucket choose kare**
* **collision → same bucket me multiple nodes**
* **equals → logical key match check**
* **linked list > 8 → tree**

# 🔹 **1. Synchronized HashMap**

### Concept:

* Agar normal HashMap thread-safe nahi → multiple threads ek saath access kare to problem hoti hai.
* Solution: Collections.synchronizedMap() ka use karke **HashMap ko thread-safe** bana sakte hain.

Map<String, Integer> map = Collections.synchronizedMap(new HashMap<>());

### Features:

1. **Thread-safe**: Haan, lekin **poora map lock ho jata hai**.
   * Ek thread read/write kar raha ho → dusre thread ko wait karna padta hai
2. **Performance slow**: Kyunki **entire map lock hota hai**
3. Iteration karte waqt bhi **synchronized block lagana padta hai**:

synchronized(map) {

for(String key: map.keySet()) {

System.out.println(key);

}

}

✅ Use case: Jab **kam threads ho aur thread-safe chahiye**

# 🔹 **2. ConcurrentHashMap**

### Concept:

* Java 5+ me introduced
* **Better thread-safe alternative** to synchronized map
* Multiple threads **simultaneously read/write** kar sakte hain **without locking whole map**

### Features:

1. **Segmented Locking (Java 8+ me bucket-level locking)**
   * Map ka **poora lock nahi hota**
   * Sirf affected bucket lock hota hai → better performance
2. **Null keys not allowed**
   * Unlike HashMap, ConcurrentHashMap **null key/value allow nahi karta**
3. Iteration safe
   * Iterator **fail-safe** hota hai → concurrent modification me exception nahi deta

# 🔹 **1. Fail-Fast Iterator**

### Concept:

* **Fail-Fast iterator** ek **normal collection (HashMap, ArrayList etc.)** me use hota hai
* Agar **iteration ke dauran collection modify** hota hai (add/remove) → **ConcurrentModificationException** throw hoti hai

### Features:

1. Works on **original collection**
2. **Does not allow structural modification** during iteration
3. Fast performance (simple pointer traverse)

### Example:

ArrayList<Integer> list = new ArrayList<>();

list.add(1);

list.add(2);

for (Integer i : list) {

list.add(3); // Runtime Exception: ConcurrentModificationException

}

✅ Collections: ArrayList, HashMap, HashSet

# 🔹 **2. Fail-Safe Iterator**

### Concept:

* **Fail-Safe iterator** **original collection ka copy** pe kaam karta hai
* Original collection me modification **allowed hai**
* Exception **throw nahi hoti**

### Features:

1. Works on **clone / copy of collection**
2. Iteration safe even if collection modified
3. Performance thoda slow (copy create karna padta hai)

### Example:

CopyOnWriteArrayList<Integer> list = new CopyOnWriteArrayList<>();

list.add(1);

list.add(2);

for (Integer i : list) {

list.add(3); // No exception

}

System.out.println(list); // [1, 2, 3, 3, 3,...]

✅ Collections: ConcurrentHashMap, CopyOnWriteArrayList, CopyOnWriteArraySet

# 🔹 **1.** final **Keyword**

### Concept:

* final ek **modifier** hai jo **constant behavior** define karta hai.
* Use: **variable, method, class** ke sath

### Features:

1. **Final Variable**
   * Value **once assign** ho gayi → **change nahi kar sakte**
   * Constant bana deta hai

final int x = 10;

x = 20; // ❌ compile-time error

1. **Final Method**
   * Method ko **override nahi kar sakte**
   * Security ya behavior fix karne ke liye use

class A {

final void show() {

System.out.println("Hello");

}

}

class B extends A {

void show() {} // ❌ compile-time error

}

1. **Final Class**
   * Class ko **inherit nahi kar sakte**
   * Example: String class

final class MyClass {}

class SubClass extends MyClass {} // ❌ error

# 🔹 **2.** finally **Block**

### Concept:

* finally **exception handling ka part** hai
* **try-catch ke sath use hota hai**
* **Always execute hota hai** → chahe exception aaye ya nahi

### Features:

1. Cleanup ka kaam karta hai (file close, DB connection close)
2. Optional, lekin useful

try {

int a = 10/0;

} catch (ArithmeticException e) {

System.out.println("Exception caught");

} finally {

System.out.println("This will always run");

}

✅ Output:

Exception caught

This will always run

# 🔹 **3.** finalize() **Method**

### Concept:

* finalize() **Object class ka method** hai
* JVM garbage collection se pehle call karta hai
* Object destroy hone se pehle cleanup karne ke liye

### Features:

1. Garbage collector call karta hai → manually call mat karo
2. Resource cleanup me use ho sakta hai
3. Deprecated in Java 9+ → use try-with-resources better hai

class Test {

protected void finalize() throws Throwable {

System.out.println("Object is garbage collected");

}

}

public class Main {

public static void main(String[] args) {

Test t = new Test();

t = null;

System.gc(); // request GC

}

}

* Output depends on GC → **not guaranteed**

# 🔹 **4. Quick Comparison**

| **Keyword** | **Type** | **Purpose** |
| --- | --- | --- |
| final | Modifier | Constant variable, prevent method override, prevent inheritance |
| finally | Block | Always execute after try/catch (cleanup) |
| finalize() | Method (Object) | Called by GC before destroying object |

💡 **Trick yaad rakhne ka:**

* **final → fix kar do**
* **finally → hamesha run kare**
* **finalize → object destroy hone se pehle cleanup**

# 🔹 **1. Transient Keyword - Concept**

* transient **Java ka keyword** hai
* **Serialization ke context me use hota hai**
* Jab koi **object serialize** hota hai, **transient variables skip ho jate hain**
* Matlab: **memory me rahenge**, **file me save nahi honge**

# 🔹 **2. Syntax**

class Student implements Serializable {

String name;

transient int age; // transient variable

}

# 🔹 **3. Example**

import java.io.\*;

class Student implements Serializable {

String name;

transient int age; // won't be serialized

Student(String name, int age) {

this.name = name;

this.age = age;

}

}

public class Test {

public static void main(String[] args) throws Exception {

Student s = new Student("Amit", 25);

// Serialize

ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("student.ser"));

oos.writeObject(s);

oos.close();

// Deserialize

ObjectInputStream ois = new ObjectInputStream(new FileInputStream("student.ser"));

Student s2 = (Student) ois.readObject();

ois.close();

System.out.println("Name: " + s2.name); // Amit

System.out.println("Age: " + s2.age); // 0 (transient not serialized)

}

}

✅ Output:

Name: Amit

Age: 0

* age transient hai → **serialization me skip ho gaya** → default value 0 aaya

# 🔹 **1. Thread Lifecycle States**

Java me **Thread ke 5 main states** hote hain:

1. **New (Created)**
2. **Runnable**
3. **Running**
4. **Waiting / Timed Waiting / Blocked**
5. **Terminated (Dead)**

# 🔹 **2. States ka Detail**

### ****1️⃣ New (Created)****

* Thread **create hua hai**, lekin **start() call nahi hua**
* Is state me thread **eligible to start** hai

Thread t = new Thread();

### ****2️⃣ Runnable****

* start() call ke baad thread **Runnable state** me chala jata hai
* JVM **thread scheduler** decide karega ki **kaun sa thread run kare**
* Note: Runnable ka matlab **ready to run**, but **not yet running**

t.start(); // Runnable state

### ****3️⃣ Running****

* Thread **CPU time milte hi running state me chala jata hai**
* run() method execute hota hai

### ****4️⃣ Waiting / Timed Waiting / Blocked****

* **Waiting:** Thread indefinitely wait kar raha hai
  + wait() method use hota hai
* **Timed Waiting:** Thread limited time ke liye wait kar raha hai
  + sleep(milliseconds), join(milliseconds)
* **Blocked:** Thread **lock acquire karne ke liye wait kar raha hai**
  + Synchronized block ke case me

### ****5️⃣ Terminated (Dead)****

* Thread **apna kaam complete karne ke baad** terminate ho jata hai
* run() method **complete hone ke baad** thread terminate

# 🔹 **3. Lifecycle Flow Diagram**

New (Created)

|

| start()

v

Runnable <----+

| |

| CPU | sleep(), wait()

v |

Running ----> Waiting / Timed Waiting / Blocked

|

| run() complete

v

Terminated

# 🔹 **1. Volatile Keyword - Concept**

* volatile ek **Java keyword** hai
* **Multi-threading context me use hota hai**
* Jab variable **volatile declare** kiya jata hai → **threads ke beech me hamesha latest value share hoti hai**

### Simple English:

Agar ek thread variable change kare → dusre thread ko turant latest value mile

# 🔹 **2. Features of Volatile**

1. **Main memory se directly read/write hota hai**
   * Thread ke CPU cache me temporary copy nahi hoti
2. **No atomicity guarantee**
   * ++ operations me race condition ho sakta hai
3. **Use for flags / status variables**
   * Example: boolean running = true;

Bilkul! Chalo **Immutable Objects** ke **advantages** ko **simple aur detail** me samjhte hain.

# 🔹 **1. Immutable Object - Concept**

* **Immutable object** → object ka **state (data) change nahi ho sakta**
* Java me **String, Integer, Boolean** immutable examples hain

### Key Rule:

Once object create ho gaya → uske fields **change nahi ho sakte**

# 🔹 **2. Advantages of Immutable Objects**

### ****1️⃣ Thread-Safe (Safe in Multi-threading)****

* Immutable objects **automatically thread-safe** hote hain
* Multiple threads use kar sakte hain **without synchronization**

String s = "Hello"; // immutable

// No thread can change value of s

### ****2️⃣ Cache/Map-friendly****

* Immutable objects **hashcode change nahi karte** → HashMap, HashSet me safe
* Example: String as key in HashMap

Map<String, Integer> map = new HashMap<>();

String key = "test";

map.put(key, 1);

// key ka value change nahi ho sakta → map safe

### ****3️⃣ Safe as Constants****

* Immutable objects ko **constant ke roop me use kar sakte**
* final + immutable = perfect constant

final String NAME = "Amit";

### ****4️⃣ Easy to Debug****

* Immutable objects **state change nahi karte** → predictable behavior
* Bugs kam aate hain → easier debugging

### ****Shallow Copy****

* Object ka **reference copy** create karta hai
* Original aur copied object **same memory location ke nested objects share karte hain**
* Matlab: **nested objects me changes dono objects me reflect honge**

### ****Deep Copy****

* Object ka **completely independent copy** create karta hai
* Original aur copied object **apne-apne memory locations use karte hain**
* Nested objects ka copy bhi create hota hai → changes **reflect nahi honge**

# 🔹 **2. Example Code**

### ****Shallow Copy****

class Address {

String city;

Address(String city) { this.city = city; }

}

class Person implements Cloneable {

String name;

Address address;

Person(String name, Address address) {

this.name = name;

this.address = address;

}

protected Object clone() throws CloneNotSupportedException {

return super.clone(); // shallow copy

}

}

public class Test {

public static void main(String[] args) throws Exception {

Address addr = new Address("Delhi");

Person p1 = new Person("Amit", addr);

Person p2 = (Person) p1.clone();

p2.address.city = "Mumbai"; // changes reflected in p1

System.out.println(p1.address.city); // Mumbai

}

}

✅ Output:

Mumbai

* Shallow copy me **nested object shared** → change reflected

### ****Deep Copy****

class PersonDeep implements Cloneable {

String name;

Address address;

PersonDeep(String name, Address address) {

this.name = name;

this.address = address;

}

protected Object clone() throws CloneNotSupportedException {

PersonDeep cloned = (PersonDeep) super.clone();

cloned.address = new Address(this.address.city); // new Address object

return cloned;

}

}

public class TestDeep {

public static void main(String[] args) throws Exception {

Address addr = new Address("Delhi");

PersonDeep p1 = new PersonDeep("Amit", addr);

PersonDeep p2 = (PersonDeep) p1.clone();

p2.address.city = "Mumbai"; // changes NOT reflected in p1

System.out.println(p1.address.city); // Delhi

}

}

✅ Output:

Delhi

* Deep copy me **nested object bhi alag copy** → original safe

## 🔹 Marker Interface Kya Hai?

**Definition:**

* **Marker Interface** ek **empty interface hoti hai** — isme koi method nahi hota.
* Ye **compiler ya JVM ko batati hai** ki jo class is interface ko implement kar rahi hai, **uske objects ke saath special behavior enable karna hai**.

**Key Points:**

1. Isme **methods nahi hote**.
2. Mostly **metadata ke liye use hota hai**.
3. JVM ya APIs check karte hain ki class ne ye interface implement kiya hai ya nahi.

## 🔹 Common Examples in Java

1. **Serializable**
   * Agar class implement kare Serializable, JVM ko pata chal jata hai ki objects **file ya network pe save (serialize) kiye ja sakte hain**.
2. **Cloneable**
   * Agar class implement kare Cloneable, JVM ko pata chal jata hai ki object **clone() method se copy kiya ja sakta hai**.
3. **Remote** (RMI)
   * Batata hai ki object **remote method invocation ke liye use ho sakta hai**.

## 🔹 Example: Marker Interface

// Marker interface

interface MyMarker {}

// Class implementing marker interface

class Employee implements MyMarker {

String name;

int id;

Employee(String name, int id) {

this.name = name;

this.id = id;

}

@Override

public String toString() {

return name + ":" + id;

}

}

// Main class

public class Main {

public static void main(String[] args) {

Employee e = new Employee("Alice", 101);

// Check if object is marked with MyMarker

if (e instanceof MyMarker) {

System.out.println("Employee object is marked with MyMarker!");

} else {

System.out.println("Employee object is not marked.");

}

}

}

**Output:**

Employee object is marked with MyMarker!

✅ Yaha MyMarker ke methods nahi hain, lekin JVM ya program check kar sakta hai instanceof se aur special behavior apply kar sakta hai.

### 🔹 Summary

| **Feature** | **Marker Interface** |
| --- | --- |
| Methods | ❌ None |
| Purpose | JVM ko batana ki class ko special treatment chahiye |
| Examples | Serializable, Cloneable, Remote |
| Use | Mostly **serialization, cloning, RMI** |

💡 **Tip:**  
Marker Interface basically **“tag” ya “flag” ka kaam karta hai**.

* Example: "Ye class serializable hai" → JVM jaanta hai aur serialize kar sakta hai.

# 🔹 **1. API Gateway - Concept**

* **API Gateway** ek **server / service** hai jo **client requests ko backend services tak route karta hai**
* Basically, **single entry point** for multiple microservices
* Clients **directly microservices ko call nahi karte**, sab request **API Gateway ke through** aati hai

### Simple Example:

Imagine a shopping app with **UserService**, **OrderService**, and **PaymentService**

* Instead of calling each service individually → client calls **API Gateway**
* Gateway forwards request to correct service, aggregates response if needed

# 🔹 **2. Responsibilities of API Gateway**

1. **Routing**
   * Request ko correct microservice me forward karna
2. **Authentication / Authorization**
   * Check user credentials before forwarding request
3. **Load Balancing**
   * Requests ko multiple service instances me distribute karna
4. **Caching**
   * Frequently used responses ko cache kar sakta hai
5. **Rate Limiting**
   * Excessive requests ko block ya throttle karna
6. **Response Aggregation**
   * Multiple microservices se data collect karke single response dena

# 🔹 **1. Concept**

### ****Service Registry****

* **Central database / repository** jahan **saari microservices ka location info (IP, port, metadata)** store hota hai
* Example: **Eureka Server, Consul, Zookeeper**
* Jaise ek **directory / phonebook** jisme har service ka address hota hai

### ****Service Discovery****

* **Mechanism** jisse microservice ya client **registry se service ka location pata karta hai**
* Basically: **“Kaunsa service kahaan hai?”**
* Do types:
  1. **Client-side discovery** → client directly registry se location fetch karke service call karta hai
  2. **Server-side discovery** → API Gateway ya Load Balancer registry se location fetch karta hai, client ko forward karta hai

# 🔹 **2. Example Analogy**

* Imagine ek **office building**
  + **Service Registry** → Reception desk me “employee list + room numbers”
  + **Service Discovery** → Aap reception se poochte “John ka room number kya hai?” → use pata chal jata hai kaunse room me hai

# 🔹 **3. Example Flow in Microservices**

Suppose we have **UserService** and **OrderService**:

UserService registers itself in Service Registry (Eureka)

OrderService registers itself in Service Registry

Client wants UserService info:

1. Client queries Service Registry

2. Gets UserService location (IP:Port)

3. Calls UserService directly OR via API Gateway

# 🔹 **4. Spring Cloud Example**

### ****1️⃣ Service Registry (Eureka Server)****

@SpringBootApplication

@EnableEurekaServer

public class EurekaServerApp {

public static void main(String[] args) {

SpringApplication.run(EurekaServerApp.class, args);

}

}

### ****2️⃣ Service Registration (UserService)****

@SpringBootApplication

@EnableEurekaClient

public class UserServiceApp {

public static void main(String[] args) {

SpringApplication.run(UserServiceApp.class, args);

}

}

**application.yml for UserService:**

eureka:

client:

service-url:

defaultZone: http://localhost:8761/eureka/

instance:

prefer-ip-address: true

* **UserService** automatically **registers itself** in Eureka Server

### ****3️⃣ Service Discovery****

@Autowired

private DiscoveryClient discoveryClient;

public String getUserServiceURL() {

List<ServiceInstance> instances = discoveryClient.getInstances("USER-SERVICE");

return instances.get(0).getUri().toString();

}

* Client ya another service **discoveryClient se location fetch** karta hai

# **1. Circuit Breaker - Concept**

* **Circuit Breaker** ek **design pattern** hai jo **microservices me failures ko handle karne ke liye** use hota hai
* Jaise **electrical circuit breaker**, jo overload ya short-circuit me **current ko rok deta hai**, waise hi ye **failing service calls ko rok deta hai**

### ****Purpose:****

1. Failing service par continuously call na bhejna → system overload na ho
2. Fast failure response → client ko immediately response mil jaye
3. Fault isolation → ek service fail ho bhi jaye to poore system ko affect na kare

# 🔹 **2. Circuit Breaker States**

1. **Closed (Normal State)**
   * Sab service calls allow hote hain
   * Agar **failure threshold** exceed ho jaye → breaker **Open**
2. **Open (Tripped / Breaker Open)**
   * All calls **block** kar diye jate hain
   * Client ko **fallback response** milta hai
   * Timeout ke baad breaker **Half-Open** me jata hai
3. **Half-Open (Test State)**
   * System test karta hai service ko
   * Agar request **successful** → breaker **Closed**
   * Agar request **fail** → breaker **Open** again

# 🔹 **3. Analogy**

* Electrical Circuit → overload → breaker trips
* Microservices → service fail → circuit breaker opens
* After timeout → try again (half-open)

# 🔹 **4. Example Flow**

Client --> Circuit Breaker --> Service

State = Closed

Call 1: Success → continue

Call 2: Fail → count++

Call 3: Fail → count exceeds threshold → Open state

State = Open

All calls blocked → return fallback response

After timeout → Half-Open

Call test request

Success → Closed

Fail → Open again

# 🔹 **1. String Constant Pool - Concept**

* **String Constant Pool (SCP)** → JVM me **special memory area inside heap**
* Purpose: **String objects ka reuse aur memory optimization**
* Jab **String literals** create hote hain → JVM **pool me check karta hai**
  + Agar same value already present → existing reference return
  + Agar new value → naya object create

Basically, JVM **duplicate String objects create hone se bachata hai**

# 🔹 **2. String Creation Types**

### ****1️⃣ Using String Literal****

String s1 = "Hello"; // stored in SCP

String s2 = "Hello"; // reference to same SCP object

System.out.println(s1 == s2); // true

* == checks reference → same object in SCP
* **Memory efficient**

### ****2️⃣ Using**** new ****Keyword****

String s3 = new String("Hello"); // creates object in heap

System.out.println(s1 == s3); // false

* new → **always heap me object create**
* SCP me existing literal use ho sakta hai
* == → false, s1.equals(s3) → true

# 🔹 **3.** intern() **Method**

* intern() → string ko **SCP me store karna / reference return karna**

String s4 = new String("Hello").intern();

System.out.println(s1 == s4); // true

* Agar SCP me value exist → reference return
* Agar exist nahi → JVM pool me add karta hai

# 🔹 **4. Memory Behavior**

| **String Creation** | **Location** | **Reference Share?** |
| --- | --- | --- |
| Literal | SCP | Yes |
| new String() | Heap | No |
| intern() | SCP | Yes (existing or added) |

* SCP helps **memory optimization**
* Especially useful in **large applications with many repeated strings**

# 🔹 **5. Key Points**

1. **Immutable Strings** → SCP me reuse safe
2. **Literal reuse** → memory efficient
3. **new keyword** → always heap object → separate reference
4. **intern()** → forces SCP storage