

✓ Java Concepts – Hindi Notes with Examples

◆ 1. Interface में Private Method

✧ Concept:

Java 9 से हम **interface** के अंदर **private methods** define कर सकते हैं।

◆ क्यों?

- Reuse के लिए (default/static methods के अंदर)
- Code duplication से बचने के लिए

□ Example:

```
interface MyInterface {
    default void display() {
        log("Showing message...");
    }

    static void printInfo() {
        logStatic("Static info...");
    }

    private void log(String msg) {
        System.out.println("Log: " + msg);
    }

    private static void logStatic(String msg) {
        System.out.println("Static Log: " + msg);
    }
}
```

◆ Note: Private methods सिर्फ interface के अंदर ही usable होते हैं।

◆ 2. Interface के Variables को Subclass में Use करना

✧ Concept:

Interface में जो भी variable होता है, वो **by default**:

- `public static final` होता है

इसलिए subclass में वो variable accessible होता है लेकिन **immutable** होता है (value change नहीं कर सकते)।

❏ Example:

```
interface MyInterface {
    int VALUE = 100; // public static final
}

class MyClass implements MyInterface {
    void show() {
        System.out.println(VALUE); // ✅ Accessible
    }
}
```

◆ Note: Interface का variable constant होता है — उसे override या change नहीं कर सकते।

◆ 3. Abstract Class में Public Variable और Subclass में उसका Use

✦ Concept:

Abstract class में आप कोई भी variable define कर सकते हैं (public, private, protected, etc). public या protected variable subclass में **directly accessible** होता है।

❏ Example:

```
abstract class Parent {
    public int number = 50;
}

class Child extends Parent {
    void show() {
        System.out.println(number); // ✅ Accessible
    }
}
```

◆ Note: Abstract class के variables instance हो सकते हैं, unlike interface के constants।

◆ 4. Same Name & Same Parameters but Different Return Type – Possible?

✦ Concept:

Java में ऐसा नहीं हो सकता कि दो methods:

- नाम same हो
- parameters same हों
- और सिर्फ return type अलग हो

✗ Example (Invalid):

```
class Test {
    int getData() {
        return 1;
    }

    String getData() {
        return "hello"; // ✗ Error: method already defined
    }
}
```

◆ Compiler method को सिर्फ नाम और parameters से distinguish करता है, return type से नहीं।

◆ 5. Method Overloading vs Covariant Return Types

✓ Method Overloading:

- नाम same
- parameters अलग

```
class Example {
    void show(int a) {}
    void show(String b) {}
}
```

✓ Covariant Return Type (Inheritance में):

- Return type subclass हो सकता है

```
class A {
    A get() { return this; }
}

class B extends A {
    @Override
    B get() { return this; } // ✓ Allowed
}
```

◆ यह सिर्फ **overriding** में allowed है, overloading में नहीं।

Aapka sawaal hai:

Agar hum do default methods ko same class mein implement karein to kya hota hai?

Iska jawab depend karta hai ke **wo default methods kahan se aa rahe hain** — agar **do alag interfaces** se aap same naam ke default method inherit karte hain, tab **ambiguity** hoti hai. Chaliye isko systematically samjhte hain.

✓ Java mein default method kya hota hai?

Java 8 se interface mein default method define karne ki suvidha mili — jisme method ka body hota hai. Ye method implementing class ko milta hai bina override kiye bhi.

◆ Case: Do interfaces mein same naam ka default method

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

interface B {
    default void show() {
        System.out.println("B's show");
    }
}

class MyClass implements A, B {
    // Compilation error aayega agar override nahi kiya
}
```

✗ Error:

class MyClass inherits unrelated defaults for show() from types A and B

● Compiler confused hota hai ki kaunsa show() method lena chahiye — isliye **ambiguity** ka error aata hai.

✓ Solution: Aapko method override karna padega

```
class MyClass implements A, B {
    @Override
    public void show() {
        // Aap chaho to kisi ek interface ka method explicitly call kar
        // sakte ho
        A.super.show(); // ya B.super.show();
    }
}
```

◆ Output:

A's show

1. Constructor Injection

Concept:

Constructor Injection mein hum dependencies ko class ke constructor ke through inject karte hain. Spring container automatically dependencies ko pass karta hai jab wo object banaata hai.

How It Works:

- Spring automatically class ke constructor ko call karta hai, aur required dependencies ko usme inject kar deta hai.
- Ye dependencies **immutable** hoti hain, matlab ek baar inject ho gayi, toh unhe modify nahi kiya jaa sakta.

Example:

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class UserService {

    private final UserRepository userRepository; // Required dependency
    private final LoggerService loggerService; // Optional dependency

    // Constructor Injection for both required and optional dependencies
    @Autowired
    public UserService(UserRepository userRepository, LoggerService
loggerService) {
        this.userRepository = userRepository;
        this.loggerService = loggerService;
    }

    public void createUser(String name) {
        if (loggerService != null) {
            loggerService.log("Creating user: " + name);
        }
        userRepository.save(name);
    }
}
```

- **Constructor Injection** mein dependencies ko constructor ke through inject kiya gaya hai. Jab **UserService** ka object banega, Spring **UserRepository** aur **LoggerService** ko inject karega.

Advantages:

1. **Immutable Dependencies:** Dependencies ko object banne ke baad change nahi kiya jaa sakta.
2. **Clear and Explicit:** Saare dependencies ek jagah constructor mein dikhenge, jo code ko clear banata hai.
3. **Easy to Test:** Constructor ke through inject ki gayi dependencies ko test karna aasan hota hai.

Disadvantages:

1. **Complexity for Optional Dependencies:** Agar koi dependency optional ho, toh constructor injection me thoda problem ho sakta hai, kyunki saari dependencies ko constructor mein pass karna padta hai.
2. **No Flexibility After Object Creation:** Ek baar object ban gaya, toh aap dependencies ko change nahi kar sakte.

Best Use Case:

- **Required Dependencies** ko inject karne ke liye.
 - Jab aap chahte ho ki dependencies immutable ho, matlab ek baar object banne ke baad unko change nahi kiya jaa sake.
-

2. Setter Injection

Concept:

Setter Injection mein dependencies ko class ke setter methods ke through inject kiya jaata hai. Yahan pe aap optional dependencies ko inject kar sakte ho.

How It Works:

- Spring class ke **setter methods** ko call karke dependencies ko inject karta hai.
- Agar koi dependency optional ho, toh aap usko **setter** ke through inject kar sakte hain.

Example:

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class UserService {

    private UserRepository userRepository; // Required dependency
    private LoggerService loggerService; // Optional dependency

    // Constructor Injection for required dependency
    @Autowired
    public UserService(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    // Setter Injection for optional dependency
    @Autowired
    public void setLoggerService(LoggerService loggerService) {
        this.loggerService = loggerService; // LoggerService ko inject kar
        rahe hain
    }

    public void createUser(String name) {
        if (loggerService != null) {
```

```

        loggerService.log("Creating user: " + name); // Logging
    }
    userRepository.save(name); // User ko save karna
}
}

```

- **Setter Injection** mein, **LoggerService** ko optional dependency ke roop mein inject kiya gaya hai. Agar **LoggerService** available hai, toh logging kiya jaayega; agar nahi hai, toh logging skip ho jaayega.

Advantages:

1. **Flexible and Configurable:** Dependencies ko baad mein bhi set kiya jaa sakta hai.
2. **Optional Dependencies:** Agar koi dependency optional ho, toh setter injection usko handle kar sakta hai.
3. **Less Boilerplate Code:** Agar constructor injection ka use kiya jaye toh, aapko optional dependencies ko handle karne mein extra methods likhne padte hain, jo setter injection se kam ho jaate hain.

Disadvantages:

1. **Mutable Dependencies:** Dependencies ko object banne ke baad bhi change kiya jaa sakta hai.
2. **Harder to Test:** Setter methods ko mock karna thoda complex ho sakta hai, aur unit testing mein challenges aa sakte hain.
3. **Hidden Dependencies:** Dependencies ko class ke setters mein dikhna thoda confusing ho sakta hai.

Best Use Case:

- **Optional Dependencies** ke liye. Agar kuch dependencies object banne ke baad set karni ho, toh setter injection best option hai.
- Jab flexibility ki zaroorat ho.

3. Field Injection

Concept:

Field Injection mein dependencies ko **directly class ke fields** mein inject kiya jaata hai. Aapko constructor ya setter method likhne ki zaroorat nahi hoti, bas **@Autowired** annotation ko field ke upar laga dena hota hai.

How It Works:

- Spring directly class ke fields mein dependencies ko inject kar leta hai. Ye approach simple aur easy hoti hai, lekin ye testing aur readability mein issues create kar sakti hai.

Example:

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class UserService {

    @Autowired // Automatically injects LoggerService into this field
    private LoggerService loggerService; // Optional dependency

    @Autowired // Automatically injects UserRepository into this field
    private UserRepository userRepository; // Required dependency

    public void createUser(String name) {
        if (loggerService != null) {
            loggerService.log("Creating user: " + name); // Logging
        }
        userRepository.save(name); // User ko save karna
    }
}
```

- **Field Injection** mein, **@Autowired** annotation **directly** class ke fields pe laga diya gaya hai, jisse Spring automatically dependencies ko inject karega.

Advantages:

1. **Minimal Code:** Aapko koi constructor ya setter method nahi likhna padta, bas **@Autowired** laga ke kaam ho jaata hai.
2. **Quick and Easy:** Dependencies ko inject karne ka simplest tareeka.

Disadvantages:

1. **Difficult to Test:** Unit testing mein issues aa sakte hain, kyunki aapko directly fields ko mock karna padta hai.
2. **Hidden Dependencies:** Jo dependencies class ke andar field ke through inject hoti hain, wo directly visible nahi hoti. Ye kabhi kabhi code ko samajhne mein mushkil kar sakta hai.
3. **Hard to Ensure Immutability:** Dependencies ko change karna aasan ho jaata hai, jo code ko unstable bana sakta hai.

Best Use Case:

- **Quick prototyping** ke liye, jab aapko jaldi se code likhna ho aur flexibility chahiye ho.
- **Simple applications** mein jahan dependencies ko directly inject kiya jaa sakta ho aur unit testing ka zaroorat na ho.

Comparison of Constructor, Setter, and Field Injection:

Feature	Constructor Injection	Setter Injection	Field Injection
Dependency Binding	Dependencies are set during object creation.	Dependencies can be changed later via setter.	Dependencies are set directly on fields.

Feature	Constructor Injection	Setter Injection	Field Injection
Immutability	Immutable dependencies (cannot be changed after object creation).	Mutable dependencies (can be changed after object creation).	Mutable dependencies (can be changed after object creation).
Ease of Testing	Easy to test (constructor parameters are clear).	Testing can be difficult due to setters.	Difficult to test due to hidden dependencies.
Best Use Case	Required dependencies, immutable state.	Optional dependencies, flexibility.	Quick prototyping, simple use cases.
Disadvantages	Difficult for optional dependencies.	Dependencies can be changed, harder to ensure immutability.	Hard to test and understand due to hidden dependencies.

Summary:

- **Constructor Injection:** Best for **required dependencies**. Clear and explicit. Dependencies are immutable, and the object is fully initialized at creation time.
 - **Setter Injection:** Ideal for **optional dependencies**. More flexible but can lead to mutable state, and harder to test and maintain.
 - **Field Injection:** Easiest and quickest way to inject dependencies but sacrifices testability and clarity. Good for **quick prototyping** or simple applications.
-

2. Bean Lifecycle in Spring

Concept:

Bean Lifecycle ka matlab hai ki Spring container ek bean ko kis tarah se create, initialize, use, aur destroy karta hai.

Lifecycle Phases:

1. **Instantiation:** Jab Spring container apne bean definition ko read karta hai, toh wo bean ka object create karta hai.
2. **Populating Properties:** Spring container bean ke properties ko **dependency injection** ke through populate karta hai.
3. **Bean Initialization:** Bean ko initialize karte waqt Spring kuch methods (like `@PostConstruct`) ko call kar sakta hai.
4. **Bean Usage:** Jab bhi koi code ya class is bean ko use karna chahti hai, Spring container wo bean provide kar deta hai.
5. **Bean Destruction:** Jab Spring container ko bean ki zaroorat nahi hoti (application shutdown), tab wo bean ko destroy kar deta hai.

Lifecycle Methods:

- **@PostConstruct:** Bean ke initialize hone ke baad is method ko call kiya जाता है.
- **@PreDestroy:** Bean destroy hone se pehle is method ko call kiya जाता है.

- **Custom Init and Destroy Methods:** Aap custom methods bhi define kar sakte ho jo bean initialization aur destruction ke time par call ho.

Example:

```
import org.springframework.stereotype.Component;
import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;

@Component
public class UserService {

    // Initialization method with @PostConstruct
    @PostConstruct
    public void init() {
        System.out.println("UserService Bean Initialized!");
    }

    // Destroy method with @PreDestroy
    @PreDestroy
    public void destroy() {
        System.out.println("UserService Bean Destroyed!");
    }

    public void createUser(String name) {
        System.out.println("User " + name + " created!");
    }
}
```

- Jab Spring container **UserService** bean ko initialize karega, toh **@PostConstruct** method ko call karega.
- Jab Spring container is bean ko destroy karega, toh **@PreDestroy** method ko call karega.

Detailed Lifecycle Example (Using @PostConstruct and @PreDestroy):

1. **Bean Creation:** Spring container **UserService** bean ka object banata hai.
2. **Dependency Injection:** Agar **UserService** ko kisi aur bean ki zaroorat hai, toh wo inject kiya jaata hai.
3. **Bean Initialization:** Spring **@PostConstruct** ko call karta hai, agar define ho.
4. **Bean Usage:** Bean ko application use karti hai jab zaroorat ho.
5. **Bean Destruction:** Application ke shutdown par **@PreDestroy** ko call kiya jaata hai, aur bean destroy ho jaata hai.

3. Bean Scope in Spring

Concept:

Bean Scope define karta hai ki Spring container ek bean ka instance kitni baar create karega. Scope ke 5 types hain, jo Spring container ke behaviour ko define karte hain.

Common Scopes:

1. **Singleton (Default):**

- **Scope:** Ek hi instance pura application mein use hota hai.
- **Usage:** Agar aap chahte hain ki application mein ek hi bean ka instance ho, toh aap singleton scope use karte hain.

Example:

```
@Component
@Scope("singleton") // Singleton is default, so this is optional
public class UserService {
    //...
}
```

Prototype Scope

Concept:

Jab Spring container ko **Prototype scope** mein ek bean ka request milta hai, toh wo **har baar** ek naya instance create karta hai. Matlab, ek bean ka **ek hi instance** ka use multiple times nahi hota. Har request par **naya object** create hota hai.

Example:

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.ApplicationContext;
import
org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.stereotype.Component;
import org.springframework.context.annotation.Scope;

@Component
@Scope("prototype") // Prototype scope: har baar naya instance create hoga
public class UserService {

    public UserService() {
        System.out.println("UserService instance created!");
    }

    public void createUser(String name) {
        System.out.println("User " + name + " created!");
    }
}

public class MainApp {

    public static void main(String[] args) {
        // Spring container ko initialize karna
        AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext(MainApp.class);

        // UserService ka 1st instance lena
        UserService userService1 = context.getBean(UserService.class);
        userService1.createUser("John");

        // UserService ka 2nd instance lena
        UserService userService2 = context.getBean(UserService.class);
```

```
        userService2.createUser("Jane");
    }
}
```

Output:

```
UserService instance created!
User John created!
UserService instance created!
User Jane created!
```

Explanation:

- **@Scope("prototype")** ka matlab hai ki Spring har baar jab **UserService** bean ka object maangega, wo ek **naya instance** create karega.
- **First Request:** Jab `context.getBean(UserService.class)` pehli baar call kiya gaya, Spring container ne ek naya **UserService** ka object banaya aur **"UserService instance created!"** print kiya.
- **Second Request:** Jab `context.getBean(UserService.class)` second time call kiya gaya, Spring ne **naya object** create kiya aur phir se **"UserService instance created!"** print kiya.

Key Points:

1. **Prototype Scope** mein, **har request** par Spring ek **naya instance** banaata hai.
2. **Singleton Scope** (jo default hota hai) mein, agar ek object bana diya gaya ho toh **wo hi instance** baar-baar use hota hai.
3. **Prototype Scope** mein, agar aap same class ka object multiple times request karte ho, toh **naye objects** milte hain.

Use Case:

- Agar aapko aise beans ki zaroorat ho jo **stateless** ho, yani unka state har instance ke liye alag ho. Jaise ki **user-specific** data ya per-request calculations.

Request Scope in Spring:

Jab aap `@Scope("request")` use karte ho, toh iska matlab hai ki Spring container **ek bean** ka **naya instance** har **HTTP request** ke liye create karega.

Key Concept:

- **Request scope** ka use web applications mein hota hai, jahan ek HTTP request ke liye ek **naya instance** of the bean chahiye hota hai.
- Agar aap **stateless** beans bana rahe ho jo ek **HTTP request-specific** data handle karte ho, toh aap `@Scope("request")` ka use karte ho.

How Request Scope Works:

- **HTTP Request:** Har baar jab ek HTTP request aati hai, toh Spring container ek naya instance create karta hai us bean ka, jo **request scope** mein defined hota hai.
- Jab HTTP request complete ho jaati hai, uss request ka **bean instance destroy** ho jaata hai.
- Yeh scope **per request** ka hota hai, yani ek hi user ke **multiple requests** ke liye alag instances.

Use Case:

Request Scope ka use tab hota hai jab aapko chahiye ki:

1. Har HTTP request ke liye ek **independent bean** ho, jisme request-specific data ho.
2. Bean ka **state** ek request ke andar ho, aur dusri request ke liye naya state ho.

Example Use Case:

Suppose aap ek **Web Application** bana rahe ho jisme har user ke liye **individual request-specific** service (jaise ki **user authentication** ya **user session management**) handle karni hai.

Detailed Example:

Maan lo hum ek **UserService** class bana rahe hain jisme **user-specific data** process hota hai, jaise user ka naam, email, ya authentication status. Har request par yeh data alag hoga, toh isko request scope mein define karna zaroori hai.

Example Code:

```
import org.springframework.context.annotation.Scope;
import org.springframework.stereotype.Component;

@Component
@Scope("request") // Request Scope: each HTTP request gets a new instance
of this bean
public class UserService {

    private String userName;

    // Constructor to set user name
    public UserService() {
        this.userName = "Anonymous User"; // Default value
    }

    // Method to set user name from the HTTP request
    public void setUserName(String userName) {
        this.userName = userName;
    }

    public String getUserName() {
        return userName;
    }

    public void processUserRequest() {
        System.out.println("Processing request for: " + userName);
    }
}
```

```
}
```

Controller (where HTTP request handling takes place):

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;

@Controller
public class UserController {

    @Autowired
    private UserService userService; // Injecting UserService

    public void handleRequest(String userName) {
        // Set user-specific name for this HTTP request
        userService.setUserName(userName);

        // Processing the request
        userService.processUserRequest();
    }
}
```

Main Application Class (to simulate HTTP request handling):

```
import org.springframework.context.ApplicationContext;
import
org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class MainApp {

    public static void main(String[] args) {
        // Initialize Spring container
        ApplicationContext context = new
        AnnotationConfigApplicationContext(MainApp.class);

        // Simulating HTTP request for two different users
        UserController controller1 = context.getBean(UserController.class);
        controller1.handleRequest("John Doe");

        UserController controller2 = context.getBean(UserController.class);
        controller2.handleRequest("Jane Smith");
    }
}
```

Output:

```
Processing request for: John Doe
Processing request for: Jane Smith
```

Explanation:

- **UserService** is defined with **@Scope("request")**. Iska matlab hai ki **UserService** ka naya instance **per HTTP request** create hoga.
- **UserController** ko **UserService** bean inject kiya gaya hai.
- Jab hum **handleRequest("John Doe")** call karte hain, toh **UserService** ka ek naya instance create hota hai, jisme **userName = "John Doe"**.

- Fir jab **handleRequest("Jane Smith")** call karte hain, toh ek aur **naya instance** create hota hai, jisme `userName = "Jane Smith"`.

Key Points:

1. **Request scope** mein, Spring har HTTP request ke liye **naya bean instance** create karta hai.
2. Ek **HTTP request** ke dauran **bean** ka state rakh sakte ho, lekin wo state **next HTTP request** ke liye retain nahi hota.
3. **Web applications** mein use hota hai jab aapko **request-specific data** handle karna ho, jaise user authentication, session data, etc.

Real-World Use Case:

1. **User Authentication:** Agar aapke paas **user login data** hai aur aapko har request mein alag **user-specific data** handle karna hai, toh **request scope** ka use karke har HTTP request ke liye **unique instance** create kar sakte ho. Isse har request ka data independent hoga.
2. **User Sessions:** Agar aap user ke session ko manage kar rahe ho aur har HTTP request par **session-specific details** ko store karna hai, toh aap **request scope** ka use kar sakte ho. Jaise ki, ek request mein user ka email, session id, aur preferences, jo agle request mein nahi chaahiye.
3. **Web Forms:** Agar aapke paas **web forms** hain jisme har user ka **form data** process ho raha hai, toh aap **request scope** ka use kar sakte ho taaki har HTTP request ka apna form state ho.

Session Scope in Spring:

Session scope ka use tab hota hai jab aapko har **HTTP session** ke liye **ek unique instance** chahiye ho. Matlab, jab user login karta hai ya koi session establish hota hai, toh aap chaahte ho ki us user ka **data** (jaise user preferences, session variables, etc.) ek specific **session** ke liye store ho aur dusre user ke session ke liye independent ho.

Concept of Session Scope:

- **Session Scope** ka matlab hai ki Spring container **ek bean ka instance** har **HTTP session** ke liye create karega.
- Jab user ka session **expire** hota hai ya **end** hota hai, toh us session ke liye jo instance create kiya gaya tha, wo destroy ho jaata hai.
- Agar user **multiple times** web application mein aata hai, toh uska **session** same rehta hai aur bean ka **same instance** us session ke liye use hota hai.

Use Case:

Session Scope ka use tab hota hai jab aapko **user-specific session data** chahiye ho. Jaise ki:

1. **User preferences** (like theme, language, etc.)
2. **Shopping cart** in e-commerce websites (for logged-in users)
3. **User authentication state** (whether the user is logged in or not)

Is case mein, har session ke liye ek naya instance hoga aur jab tak user ka session active rahega, tab tak us instance ka state maintained rahega.

Example of Session Scope:

Step 1: Define a Session Scoped Bean (e.g., UserSession)

```
import org.springframework.context.annotation.Scope;
import org.springframework.stereotype.Component;

@Component
@Scope("session") // Session Scope: one instance per HTTP session
public class UserSession {

    private String username;
    private String userRole;

    public UserSession() {
        // Default constructor
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getUsername() {
        return username;
    }

    public void setUserRole(String userRole) {
        this.userRole = userRole;
    }

    public String getUserRole() {
        return userRole;
    }

    public void displaySessionInfo() {
        System.out.println("User Session - Username: " + username + ",
Role: " + userRole);
    }
}
```

Step 2: Define a Controller to Handle Requests (simulating web interaction)

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;

@Controller
public class UserSessionController {
```



```

@Autowired
private UserSession userSession;

// Simulating setting session data
public void setSessionData(String username, String userRole) {
    userSession.setUsername(username);
    userSession.setUserRole(userRole);
}

// Display session information
public void showSessionInfo() {
    userSession.displaySessionInfo();
}
}

```

Step 3: Simulate HTTP Sessions in the Main Class

```

import org.springframework.context.ApplicationContext;
import
org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class MainApp {

    public static void main(String[] args) {
        // Initializing Spring context
        ApplicationContext context = new
AnnotationConfigApplicationContext(MainApp.class);

        // Simulating two different user sessions (with different
        usernames)
        UserSessionController controller1 =
context.getBean(UserSessionController.class);
        controller1.setSessionData("john.doe", "admin");
        controller1.showSessionInfo();

        // Simulating second user session
        UserSessionController controller2 =
context.getBean(UserSessionController.class);
        controller2.setSessionData("jane.smith", "user");
        controller2.showSessionInfo();
    }
}

```

Output:

```

User Session - Username: john.doe, Role: admin
User Session - Username: jane.smith, Role: user

```

Explanation:

1. **UserSession Bean** is marked with `@Scope("session")`, meaning Spring will create a new instance of this bean for each **HTTP session**.
2. **Controller** simulates two different user sessions.
 - **controller1** handles `john.doe` (admin role).
 - **controller2** handles `jane.smith` (user role).

3. When each controller method (`setSessionData()` and `showSessionInfo()`) is called, Spring ensures that each session maintains **separate state** for `UserSession`.
 - **Session 1** (for `john.doe`) maintains `username="john.doe"` and `role="admin"`.
 - **Session 2** (for `jane.smith`) maintains `username="jane.smith"` and `role="user"`.
 4. This ensures that the two users, even though they interact with the same application, are completely isolated from each other, and their session-specific data is stored separately.
-

Use Case in Real-World Applications:

1. **User Authentication:**
 - **Session scope** can be used to store **user credentials** and authentication details once a user logs in. This ensures that the user remains logged in for the duration of their session and can access protected resources.
2. **Shopping Cart in E-commerce:**
 - In an e-commerce website, you may store the **shopping cart** in a session so that the user can continue shopping, and the items they add to their cart will be preserved throughout their session.
3. **User Preferences:**
 - If your application allows users to set preferences (like **theme, language** or **notifications**), you can use session scope to store this data and make sure it's available throughout the user's session.
4. **Stateful Web Services:**
 - If you're building a **stateful** web service, where the state of each client session needs to be maintained independently, session scope ensures that each session gets a separate stateful bean.

Chalo ab **Global Session Scope** ke baare mein baat karte hain! Yeh bhi ek important scope hai, jo kuch special cases mein kaam aata hai.

Global Session Scope in Spring:

Global Session Scope ka use tab hota hai jab aapko **web applications** mein multiple **portlets** ke liye **session-specific data** maintain karni ho. Yeh scope **Portlet-based applications** mein use hota hai, jahan ek user ka data ek **global session** mein share hota hai across multiple portlets.

Portlet basically ek mini-web application hota hai, jo ek **larger web page** mein embed hota hai. Yeh generally **portlet containers** (like Liferay) mein use hota hai.

Key Points:

1. **Global Session Scope** mein Spring container **bean ka ek instance** create karta hai, jo **poore global session** ke liye shared hota hai.
2. Is scope ka use tab hota hai jab aapko **multiple portlets** mein **shared data** chahiye ho, jo har portlet ke liye accessible ho.

How It Works:

- **Global session** ka concept primarily **portlet containers** mein use hota hai, jahan **multiple portlets** ek hi session ke andar ek dusre ke data ko access kar sakte hain.
- Is scope mein ek **shared instance** hota hai across all portlets of the global session.

Use Case:

- Agar aapke paas **multiple portlets** hain, aur aap chahte hain ki ek portlet ka state ya data doosre portlet ke saath **share** ho, toh aap **Global Session Scope** use kar sakte ho.
- Jaise, agar ek portlet user ka profile display kar raha hai aur doosra portlet user ke orders show kar raha hai, toh dono portlets ko ek hi user-specific data chahiye ho sakta hai. Yeh data **global session** mein store kiya jaata hai.

Java Exception Handling - Simple Notes

1. What is an Exception?

- **Exception** ek **unexpected** event hai jo program ke execution ko interrupt karta hai.
- Jab kuch **galat** hota hai, toh exception hoti hai.

2. Types of Exceptions

- **Checked Exceptions:**
 - Java **compile-time** pe inhe check karta hai.
 - **Mandatory** hai inhe handle karna.
 - **Example:** `IOException`, `SQLException`, `FileNotFoundException`.
 - **Unchecked Exceptions:**
 - Java **runtime** pe inhe check karta hai.
 - **Optional** hai inhe handle karna.
 - **Example:** `ArithmeticException`, `NullPointerException`, `ArrayIndexOutOfBoundsException`.
-

3. Common Exceptions

- **Checked Exceptions:**
 - **IOException:** Input/output errors jaise file read/write issues.
 - **SQLException:** Database-related issues.
 - **FileNotFoundException:** Jab file nahi milti.
- **Unchecked Exceptions:**
 - **NullPointerException:** Jab aap `null` object ke method ko call karte hain.
 - **ArithmeticException:** Agar zero se divide karne ki koshish karo.
 - **ArrayIndexOutOfBoundsException:** Agar array ke limit se bahar access karne ki koshish karo.

4. Exception Handling in Java

- Java mein exception handling ke liye **try-catch** block use hota hai.
 - **try** block mein wo code likhte hain jahan exception aane ka chance ho.
 - Agar exception aati hai toh wo **catch** block mein handle hoti hai.
-

5. Syntax for Exception Handling

```
try {  
    // Code that might cause an exception  
} catch (ExceptionType e) {  
    // Handling exception here  
}
```

Example:

```
public class Example {  
    public static void main(String[] args) {  
        try {  
            int result = 10 / 0; // Will throw ArithmeticException  
        } catch (ArithmeticException e) {  
            System.out.println("Error: Cannot divide by zero!");  
        }  
    }  
}
```

6. throws VS throw

- **throw**: Jab aap manually exception ko throw karte ho.
 - **Usage**: Jab aapko kisi specific condition pe exception manually throw karni ho.

Example:

```
public class Example {  
    public static void main(String[] args) {  
        throw new IllegalArgumentException("Invalid input!");  
    }  
}
```

- **throws**: Jab aap method ke signature mein batate ho ki yeh method exception throw kar sakta hai.
 - **Usage**: Agar aapka method checked exception throw kare, toh aapko **throws** use karna padta hai.

Example:

```
public class Example {
```

```
        public static void main(String[] args) throws IOException {
            readFile();
        }

        public static void readFile() throws IOException {
            // Some code that might throw IOException
        }
    }
}
```

7. Custom Exceptions

- Aap apni khud ki exception create kar sakte ho jo **Exception** ya **RuntimeException** class ko inherit kare.

Example:

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

public class Example {
    public static void main(String[] args) throws AgeException {
        int age = 15;
        if (age < 18) {
            throw new AgeException("Age must be 18 or above!");
        }
    }
}
```

8. Exception Propagation

- Agar method mein exception handle nahi hoti, toh wo exception calling method tak **propagate** ho jaati hai.

Example:

```
public class Example {
    public static void main(String[] args) {
        try {
            method1();
        } catch (Exception e) {
            System.out.println("Handled in main: " + e.getMessage());
        }
    }

    public static void method1() throws Exception {
        method2(); // Propagating exception to method2
    }

    public static void method2() throws Exception {
        throw new Exception("Something went wrong!");
    }
}
```

9. Best Practices for Exception Handling

- **Handle only expected exceptions:** Sirf un exceptions ko handle karo jo tum predict kar sakte ho.
 - **Don't use exceptions for normal flow:** Exception ko normal program flow ke liye mat use karo.
 - **Provide useful messages:** Exception messages clear aur useful hone chahiye.
 - **Log exceptions:** Exception ke details ko log karna chahiye taaki debugging aasaan ho.
-

10. Summary

- **Exception** kisi bhi error ko handle karne ka tarika hai jo program ki execution ko rok sakti hai.
 - **Checked exceptions** ko **mandatory** handle karna padta hai.
 - **Unchecked exceptions** ko handle karna optional hai, lekin handling best practice hai.
 - **throw** se exception manually throw karte hain, aur **throws** se method declare karte hain ki yeh exception throw karega.
-

In Short:

1. **Checked Exception:** Compile-time pe check hoti hai, handle karna zaroori hota hai (use `throws`).
2. **Unchecked Exception:** Runtime pe check hoti hai, handle karna optional hota hai.
3. **throw:** Manual exception throw karna.
4. **throws:** Method ke signature mein declare karna ki yeh method exception throw karega.
- 5.

Here's the simplified **Java Exception Hierarchy** with arrows:

```
Throwable
|
|-- Error
|   |-- OutOfMemoryError
|   |-- StackOverflowError
|
|-- Exception
|   |-- Checked Exceptions
|       |-- IOException
|       |-- SQLException
|       |-- FileNotFoundException
|   |-- RuntimeException
|       |-- NullPointerException
```

```
-- ArithmeticException
-- ArrayIndexOutOfBoundsException
```

Explanation:

1. **Throwable:** The top-level class from which all exceptions and errors derive.
2. **Error:** Critical errors that are typically not handled, such as `OutOfMemoryError` or `StackOverflowError`.
3. **Exception:** The base class for exceptions that we can handle.
 - o **Checked Exceptions:** These must be caught or declared (like `IOException`, `SQLException`).
 - o **Unchecked Exceptions (RuntimeException):** These can occur during runtime (like `NullPointerException`, `ArithmeticException`).

Question 18: Catching Throwable

```
try {
    int a = 5 / 0;
} catch (Throwable t) {
    System.out.println("Caught throwable");
}
```

Q: Kya ye sahi practice hai Throwable ko catch karna? Kya isme koi problems ho sakti hain? Kya Throwable catch karne se sari problems handle ho jayengi? Explain karo.

Throwable ko catch karne ke issues:

1. **Throwable me Error bhi aata hai** — jaise `OutOfMemoryError`, `StackOverflowError`.
Ye errors serious system-level problems hote hain jinko catch kar ke recover karna mushkil ya unsafe hota hai.
2. **Catch karne se program state unpredictable ho sakti hai.**
Agar aap `Error` catch karte hain, to JVM ki critical state ho sakti hai compromised, aur ye aapke program ko unstable bana sakta hai.
3. **Debugging difficult hoti hai.**
Kyunki aap ne sab kuch catch kar liya, kabhi kabhi pata nahi chalta ki asli problem kya thi.
4. **Best practice ye hai:**
 - o Specific exceptions (`Exception` aur uske subclasses) ko catch karo.
 - o Errors ko JVM pe chhod do, taki wo apne aap handle kare.





🔗 Notes: Redis Caching in Spring Boot Microservices

🔗 1. Redis क्या है?

- Redis = **Remote Dictionary Server**

- एक **in-memory data store** है → super-fast data access.
- Use cases:
 - Caching
 - Session storage
 - Leaderboards
 - Pub/Sub systems

2. Redis Caching क्यों?

Benefit	Description
 Performance Boost	DB calls reduce, app faster
 Memory-Based	Low latency
 Frequent Data	Hot data को बार-बार DB से fetch करने से बचना
 TTL Support	Auto-expiry possible

4. Configuration (application.yml)

```
spring:
  redis:
    host: localhost
    port: 6379
    timeout: 60000
  cache:
    type: redis

cache:
  ttl:
    user: 300 # seconds (5 min)
```

5. Enable Caching (@EnableCaching)

```
@Configuration
@EnableCaching
public class RedisConfig {

    @Bean
    public RedisCacheConfiguration cacheConfiguration() {
        return RedisCacheConfiguration.defaultCacheConfig()
            .entryTtl(Duration.ofMinutes(5))

        .serializeValuesWith(RedisSerializationContext.SerializationPair.fromSerializer(new GenericJackson2JsonRedisSerializer()));
    }
}
```

6. Entity (Serializable होना चाहिए)

```
@Data
@AllArgsConstructor
```



```
@NoArgsConstructor
public class User implements Serializable {
    private Long id;
    private String name;
    private String email;
}
```

7. Cacheable Service Layer

```
@Service
public class UserService {

    @Cacheable(value = "users", key = "#id")
    public User getUserById(Long id) {
        // DB call
    }

    @CachePut(value = "users", key = "#user.id")
    public User updateUser(User user) {
        // Update DB + cache
    }

    @CacheEvict(value = "users", key = "#id")
    public void evictCache(Long id) {
        // Remove cache
    }
}
```

8. RedisTemplate क्या है?

Feature	Description
Low-Level API	Full Redis control
Custom TTL per key	Yes
Advanced operations	Hashes, Lists, Pub/Sub, etc.
Use when	Fine control needed

Basic Usage

```
@Autowired
private RedisTemplate<String, Object> redisTemplate;

redisTemplate.opsForValue().set("user:1", userObj, Duration.ofMinutes(5));
User user = (User) redisTemplate.opsForValue().get("user:1");
redisTemplate.delete("user:1");
```

9. RedisTemplate vs @Cacheable

Feature	@Cacheable	RedisTemplate
Simplicity	 Easy	 Manual

Feature	@Cacheable	RedisTemplate
TTL Control	✗ Limited	✓ Full
Complex data types	✗ No	✓ Yes
Serialization	Auto-handled	Customizable
Best For	80% use-cases	Advanced cases

Apache Kafka – Notes (Hindi) | Basic to Deep

1. Kafka Kya Hai?

- Kafka ek **distributed messaging system** hai jo **high-throughput, scalable, aur fault-tolerant** data streaming ke liye use hota hai.
 - Real-time data pipelines aur event streaming apps banane ke liye perfect hai.
-

2. Kafka Ke Main Components

Component	Kaam
Producer	Data generate karke Kafka ko bhejne wala
Broker	Kafka ka server, jahan data store hota hai
Topic	Messages ka logical channel/category
Partition	Topic ke andar data ka ordered sub-division
Consumer	Kafka se data read karne wala
Consumer Group	Multiple consumers ka group, jo messages share karta hai
ZooKeeper	Cluster coordination aur metadata manage karta hai (Kafka 2.x tak)

3. Kafka Ki Working Process (Step by Step)

1. **Producer** data generate karta hai aur message ko **topic ke partition** me bhejta hai.
 2. **Kafka broker** messages ko partitions me store karta hai; har message ka unique **offset** hota hai.
 3. **Consumer** topic ke partitions se message read karta hai.
 4. **Consumer** apne offsets commit karta hai, taaki pata chale kaha tak data process ho chuka hai.
-

4. Partition Kya Hai Aur Kyun Zaroori Hai?

- Topic ko multiple **partitions** me todna scalability aur performance ke liye hota hai.
 - Partition ki wajah se **parallel processing** possible hoti hai.
 - Kafka **partition ke andar hi message order guarantee karta hai**.
 - Partition se **load balancing** hota hai: multiple consumers ek topic ko efficiently read kar sakte hain.
 - Partition me **replication** hota hai jisse fault tolerance aata hai.
-

5. Consumer Group Ka Role

- Multiple consumers ko ek group me daal kar workload share karte hain.
 - Har partition ka data ek hi consumer ko milta hai us group ke andar.
 - Isse **load balancing** hoti hai aur horizontally scale karna easy hota hai.
-

6. Consumer Group Aur Microservices Ka Setup (Example)

Scenario: Same group me multiple microservices

- Agar tum 3 microservices ko **same consumer group** me rakhte ho, to Kafka har partition ka data sirf ek microservice ko dega.
- Result: Har microservice ko **poora data nahi milega**; messages partition-wise divide honge.

Solution: Alag consumer groups

Microservice	Consumer Group ID
OrderProcessorService	order-group
InventoryService	inventory-group
AnalyticsService	analytics-group

- Isse har microservice ko poora topic ka data milega.

Multiple Instances of same microservice

- Multiple instances ko same **consumer group** me rakhte hain taaki messages partition-wise distribute ho jaye.
-

7. Kafka Ka Simple Analogy

- Producer = Letter writer
 - Broker/Topic = Post office box
 - Partition = Box ke alag pigeonholes
 - Consumer = Letter reader
 - Consumer group = Group of readers sharing boxes
-

8. Kafka Configuration Key Points

Config	Description
<code>bootstrap.servers</code>	Kafka brokers ke addresses
<code>group.id</code>	Consumer group ka unique ID
<code>key.serializer</code>	Producer key ko byte me convert karta hai
<code>value.serializer</code>	Producer value ko byte me convert karta hai
<code>key.deserializer</code>	Consumer key ko original form me convert karta hai
<code>value.deserializer</code>	Consumer value ko original form me convert karta hai
<code>auto.offset.reset</code>	Consumer naye group se start kare to kaha se read kare

9. Summary Table – Consumer Group Aur Message Distribution

Condition	Result
Same consumer group	Messages partition-wise distribute; ek message sirf ek consumer ko milega
Different consumer groups	Har group ko poora topic ka data milega

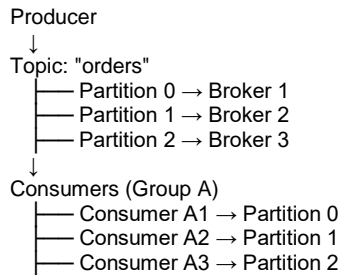
10. Kafka Use Cases

- **E-commerce:** Order placed → Kafka topic → Inventory & Billing services consume karen.
 - **Log Aggregation:** Applications ke logs Kafka me → Analytics consume karta hai.
 - **Real-time Analytics:** Streaming data ko real time process karna.
-

11. Quick Tips to Remember




- **Partitioning** se high throughput aur scalability milti hai.

- **Consumer groups** se load balancing aur multiple independent consumers bante hain.
- **Alag consumer groups** har microservice ko poora data dene ke liye zaroori hain.
- Offset management se message processing track hota hai.



Use Case: 10,000 Students ka Data — Kafka Architecture

Scenario:

- Tumne **10,000 students ki list** Kafka topic me bheji hai (via Producer)
- Alag-alag microservices ko **ye data chahiye for processing**:
 -  **Fee Service**
 -  **Physical Verification Team**
 -  **Library Team**

- ✓ **Sabko poora data chahiye**, apne respective processing ke liye
- ✓ Har team ka microservice **multiple instances me deployed** hai for load balancing

Kafka Architecture Design

Step 1: Kafka Topic

Topic: student-events
Partitions: 6 (example)

6 partitions banaye jaye taaki parallel processing ho sake.

Step 2: Producer

```

for (Student student : studentsList) {
    kafkaTemplate.send("student-events", student.getId(), student);
}
  
```

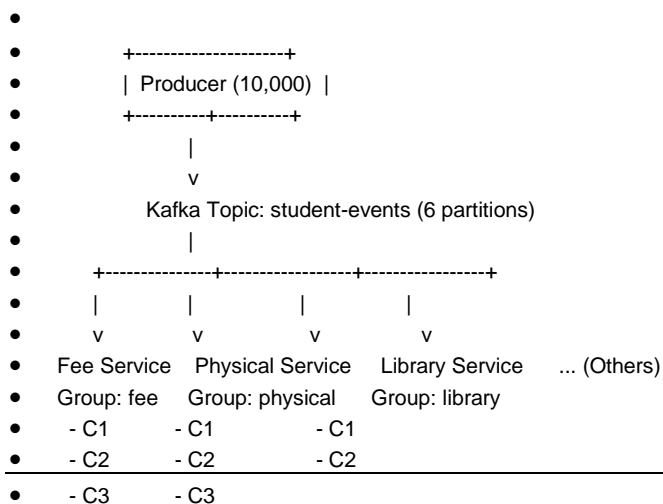
- **Key:** studentId
 - Isse same student ke saare events same partition me jaayenge (ordering maintained)
-

✦ Step 3: Microservices Setup (Consumer Groups)

Microservice	Consumer Group ID	Instances (Consumers)
Fee Service	fee-group	3 instances
Physical Verification	physical-group	3 instances
Library Service	library-group	2 instances

🔗 Important:

- Har microservice ka alag consumer group hona chahiye, tabhi sabko **poora data milega**
- Kafka **per group ek hi copy of data send karta hai**
→ To agar 3 services same group me hongi to data **divide ho jayega, repeat nahi hoga**



Functional Interface

✓ 1. Predicate<T>

✦ Purpose:

Input lo → true / false return karo
(Used for **filtering** / **conditions**)

✓ Method:

```
boolean test(T t);
```

✓ Example:

```
Predicate<String> isLong = str -> str.length() > 5;  
System.out.println(isLong.test("Hello"));    // false  
System.out.println(isLong.test("Welcome"));  // true
```

✓ 2. Consumer<T>

✚ Purpose:

Input lo → **kuch karo** (print, save, log) → kuch return mat karo
(Used for **side effects**)

✓ Method:

```
void accept(T t);
```

✓ Example:

```
Consumer<String> printer = str -> System.out.println("Hello, " + str);  
printer.accept("ChatGPT"); // Output: Hello, ChatGPT
```

✓ 3. Supplier<T>

✚ Purpose:

Kuch return karo → koi input nahi
(Used for **data suppliers**, caching, factory methods)

✓ Method:

```
T get();
```

✓ Example:

```
Supplier<Double> randomValue = () -> Math.random();  
System.out.println(randomValue.get()); // e.g., 0.7362
```

✓ 4. Function<T, R>

✚ Purpose:

Input lo → output do
(Used for **mapping, transformation**)

✓ **Method:**

```
R apply(T t);
```

✓ **Example:**

```
Function<String, Integer> stringLength = str -> str.length();  
System.out.println(stringLength.apply("Java")); // Output: 4
```

✓ 5. BiPredicate<T, U>

🔑 **Purpose:**

2 inputs lo → boolean return karo

✓ **Method:**

```
boolean test(T t, U u);
```

✓ **Example:**

```
BiPredicate<String, Integer> isLengthEqual = (str, len) -> str.length() ==  
len;  
System.out.println(isLengthEqual.test("Chat", 4)); // true
```

✓ 6. BiConsumer<T, U>

🔑 **Purpose:**

2 inputs lo → kuch kaam karo (kuch return nahi)

✓ **Method:**

```
void accept(T t, U u);
```

✓ **Example:**

```
BiConsumer<String, Integer> printInfo = (name, age) ->  
    System.out.println(name + " is " + age + " years old");  
  
printInfo.accept("Ravi", 25);
```

✓ 7. BiFunction<T, U, R>

✚ Purpose:

2 inputs lo → ek output do

✓ Method:

```
R apply(T t, U u);
```

✓ Example:

```
BiFunction<Integer, Integer, Integer> add = (a, b) -> a + b;  
System.out.println(add.apply(5, 3)); // Output: 8
```

✓ 8. UnaryOperator<T> (Special type of Function)

✚ Purpose:

Input aur output same type ka ho

✓ Method:

```
T apply(T t);
```

✓ Example:

```
UnaryOperator<String> toUpper = str -> str.toUpperCase();  
System.out.println(toUpper.apply("java")); // Output: JAVA
```

✓ 9. BinaryOperator<T> (Special type of BiFunction)

✚ Purpose:

2 inputs same type ke lo → same type ka output do

✓ Method:

```
T apply(T t1, T t2);
```

✓ Example:

```
BinaryOperator<Integer> multiply = (a, b) -> a * b;  
System.out  
.println(multiply.apply(4, 5)); // Output: 20
```

Updated Comparison Table — With Time Complexity

Feature	ArrayList	LinkedList	Vector	Stack
Structure	Dynamic Array	Doubly Linked List	Dynamic Array	Vector-based
Access Speed	✓ Fast ($O(1)$)	⊖ Slow ($O(n)$)	✓ Fast ($O(1)$)	✓ Fast ($O(1)$)
Insert at End	✓ Fast (Amortized $O(1)$)	✓ Fast ($O(1)$)	✓ Fast ($O(1)$)	✓ Fast ($O(1)$)
Insert in Middle	⊖ Slow ($O(n)$)	✓ Medium ($O(n)$)	⊖ Slow ($O(n)$)	⊖ Not Used (N/A)
Delete at End	✓ Fast ($O(1)$)	✓ Fast ($O(1)$)	✓ Fast ($O(1)$)	✓ Fast ($O(1)$)
Delete in Middle	⊖ Slow ($O(n)$)	✓ Medium ($O(n)$)	⊖ Slow ($O(n)$)	⊖ Not Used (N/A)
Search	✓ Fast ($O(n)$)	⊖ Slow ($O(n)$)	✓ Fast ($O(n)$)	✓ Fast ($O(n)$)
Thread-safe	✗ No	✗ No	✓ Yes (synchronized)	✓ Yes (inherited)
Nulls Allowed	✓ Yes	✓ Yes	✓ Yes	✓ Yes
Use-case	Random Access	Frequent Insert/Delete	Legacy, thread-safe	LIFO (Last-In-First-Out) Stack

Set vs Array in Java – Full Comparison

Feature / Point	Array	Set
Type	Data structure	Interface in Java Collection Framework
Size	Fixed (at creation)	Dynamic (grows/shrinks automatically)
Duplicates Allowed	✓ Yes	✗ No (all elements are unique)
Order Maintained	✓ Yes (Indexed, fixed order)	✗ Not guaranteed (depends on implementation)
Index-Based Access	✓ Yes (array[0], array[1], etc.)	✗ No direct index access
Null Values Allowed	✓ Yes	✓ Yes (Only one null in HashSet)
Type of Elements	Primitive & Objects (int[], String[])	Only Objects (Set<Integer>, not int)

Feature / Point	Array	Set
Resizing	✗ Not allowed (manually with new array)	✓ Auto resize
Performance	✓ Fast (no overhead)	✓ Efficient lookup (in <code>HashSet</code>)
Thread-Safety	✗ Not thread-safe	✗ By default; can be synchronized
Belongs to	Core Java	Java Collections Framework

Java Set Implementations Comparison

Feature	HashSet	LinkedHashSet	TreeSet
Ordering	✗ No order (Unordered)	✓ Insertion order maintained	✓ Sorted (natural or custom)
Duplicates Allowed	✗ No	✗ No	✗ No
Underlying Data Structure	Hash Table	Hash Table + Linked List	Red-Black Tree (Self-balancing BST)
Null Elements	✓ One null allowed	✓ One null allowed	⊖ Not allowed (throws <code>NullPointerException</code>)
Performance (Search, Insert, Delete)	✓ Fast ($O(1)$ average)	✓ Slightly slower than <code>HashSet</code>	⊖ Slower ($O(\log n)$)
Thread-safe?	✗ No	✗ No	✗ No
Sorted?	✗ No	✗ No	✓ Yes
Use-case	Fast lookup, no order	Maintain order + uniqueness	Sorted unique data

Bonus: Convert Between Array and Set

✓ Array → Set:

```
String[] arr = {"A", "B", "A"};
Set<String> set = new HashSet<>(Arrays.asList(arr));
```

✓ Set → Array:

```
Set<String> set = Set.of("A", "B");
String[] arr = set.toArray(new String[0]);
```

Examples:

HashSet Example:

```
Set<String> set = new HashSet<>();
set.add("Banana");
set.add("Apple");
set.add("Mango");
System.out.println(set); // Output order is NOT guaranteed
```

LinkedHashSet Example:

```
Set<String> set = new LinkedHashSet<>();
set.add("Banana");
set.add("Apple");
set.add("Mango");
System.out.println(set); // Output: [Banana, Apple, Mango] (insertion order)
```

TreeSet Example:

```
Set<String> set = new TreeSet<>();
set.add("Banana");
set.add("Apple");
set.add("Mango");
System.out.println(set); // Output: [Apple, Banana, Mango] (sorted order)
```




Java Map Interface – Explained with Comparison

Map क्या है?

- Map एक **key-value pair** data structure है।
- हर key unique होती है, और हर key से एक value जुड़ी होती है।
- Example: जैसे कि एक आदमी का नाम और उसकी salary

```
Map<String, Integer> salaryMap = new HashMap<>();
salaryMap.put("Ram", 50000);
salaryMap.put("Shyam", 60000);
```

Common Implementations of Map

Implementation	Ordered?	Sorted?	Allows null?	Thread-safe?
HashMap	✗ No order	✗ No	 One null key, multiple null values	✗ No
LinkedHashMap	 Insertion order	✗ No	 Yes	✗ No

Implementation	Ordered?	Sorted?	Allows null?	Thread-safe?
TreeMap	☑ Sorted by keys	☑ Yes (Natural/custom)	⊖ No null keys (throws error)	✗ No
Hashtable	✗ No order	✗ No	⊖ No null keys/values	☑ Yes (Legacy)
ConcurrentHashMap	✗ No order	✗ No	⊖ No null keys/values	☑ Yes (Modern)

🔗 Comparison Table – HashMap VS LinkedHashMap VS TreeMap

Feature	HashMap	LinkedHashMap	TreeMap
Ordering	✗ Unordered	☑ Maintains insertion order	☑ Sorted by keys
Performance	☑ Fastest (O(1))	☑ Fast (O(1))	⊖ Slower (O(log n))
Null Key	☑ 1 allowed	☑ 1 allowed	⊖ Not allowed
Null Values	☑ Allowed	☑ Allowed	☑ Allowed
Thread-safe?	✗ No	✗ No	✗ No
Internal Structure	Hash table	Hash table + LinkedList	Red-black tree (BST)

🔍 Examples

☑ 1. HashMap

```
Map<String, Integer> map = new HashMap<>();
map.put("Ram", 100);
map.put("Shyam", 200);
map.put("Ram", 150); // Overwrites
System.out.println(map); // Unordered
```

Output (unordered):

```
{Shyam=200, Ram=150}
```

☑ 2. LinkedHashMap

```
Map<String, Integer> map = new LinkedHashMap<>();
map.put("Ram", 100);
map.put("Shyam", 200);
map.put("Mohan", 300);
System.out.println(map); // Insertion order
```

Output:

```
{Ram=100, Shyam=200, Mohan=300}
```

✓ 3. TreeMap

```
Map<String, Integer> map = new TreeMap<>();  
map.put("Ram", 100);  
map.put("Shyam", 200);  
map.put("Mohan", 300);  
System.out.println(map); // Sorted by keys (alphabetically)
```

Output:

```
{Mohan=300, Ram=100, Shyam=200}
```

HashMap का Detailed Working & Concepts – Step by Step

1 Hashing क्या है?

- Hashing एक technique है जिससे हम किसी key को एक number (hash code) में convert करते हैं।
 - ये number बहुत बड़ा या छोटा हो सकता है, इसलिए इसे HashMap के array के size में फिट करने के लिए index में convert करना पड़ता है।
 - इस index पर हम value को स्टोर करते हैं।
-

2 HashMap के अंदर क्या होता है?

- HashMap में एक **array** होता है, जिसे हम **buckets** कहते हैं।
 - हर bucket में entries होती हैं (initially empty)।
 - Entries वो होते हैं जो key-value pairs को स्टोर करते हैं।
-

3 Step-by-Step put() operation

माना कि हम map में "Apple" -> 10 डाल रहे हैं:

Step 1: hashCode () निकालना

- "Apple".hashCode () निकाला जाता है, जो एक integer होता है।

```
int hash = key.hashCode();
```

Step 2: Index निकालना

- Hash को array के size से match करने के लिए index निकालते हैं:

```
int index = (hash & (n - 1));
```

- जहाँ n bucket array की length होती है (default 16 होती है शुरुआत में)।
-

Step 3: Bucket पर जाएँ और Entry डालें

- अगर उस index पर कोई और entry नहीं है, तो new entry रख दी जाती है।
 - अगर वहाँ entries पहले से हैं, तो:
 - पहले keys की तुलना करते हैं (equals method से)।
 - अगर कोई existing key है तो value update करें।
 - नहीं तो नए entry को linked list या tree में add करें।
-

Step 4: Resize की जांच

- अगर entries की संख्या `load factor * capacity` से ज्यादा हो जाए, तो HashMap का size double हो जाता है और सारी entries को फिर से redistribute (rehash) किया जाता है।
-

4 Collision Handling

Collision तब होता है जब दो अलग-अलग keys का hash index same हो।

- Java 7 तक: Collision को linked list में handle किया जाता था।
 - Java 8 से: अगर एक bucket में entries 8 से ज्यादा हो जाएं तो linked list को balanced tree (Red-Black Tree) में convert कर दिया जाता है ताकि lookup तेज हो जाए।
-

5 get() operation का flow

get("Apple") करने पर:

1. "Apple" का hashCode() निकालो।
 2. Index निकालो।
 3. उस bucket में जाकर linked list या tree traverse करो।
 4. हर entry की key से "Apple" compare करो (equals() से)।
 5. Match मिलने पर value return करो।
-

6 Example with collisions

मान लीजिए keys "Aa" और "BB" का hashCode Java में same हो सकता है (testing purpose):

```
System.out.println("Aa".hashCode()); // 2112
System.out.println("BB".hashCode()); // 2112
```

- दोनों का hash code same होने से वो same bucket में जाएंगे।
- फिर linked list या tree में stored होंगे।

7 Visual Representation

HashMap Array (size 16):

Index 5:
["Aa"=10] -> ["BB"=20] -> null (linked list due to collision)

Index 8:
["Apple"=30] -> null

8 Java 8 Optimization – Treeify

- अगर कोई bucket में 8 से ज्यादा nodes हों तो linked list को tree में convert कर दिया जाता है ताकि time complexity $O(\log n)$ हो जाए।

9 Important Parameters

Parameter	Default Value	Description
Initial Capacity	16	Bucket array की starting size
Load Factor	0.75	Resize trigger threshold (75% full)

HashMap vs ConcurrentHashMap – Detailed Comparison

Feature	HashMap	ConcurrentHashMap
Thread Safety	✗ Thread-safe नहीं, synchronized नहीं	✓ Thread-safe, internal synchronization करता है
Locking Mechanism	No locking (unsafe in multithread)	Uses Segment-level locks (Java 7) या CAS + lock-free (Java 8+) for concurrency
Null Keys & Values	✓ Allows one null key and multiple null values	✗ Null keys और null values allowed नहीं हैं

Feature	HashMap	ConcurrentHashMap
Performance (Multithreaded)	Poor — race conditions हो सकते हैं	High-performance concurrent operations, less blocking
Iteration Behavior	Fail-fast (ConcurrentModificationException)	Weakly consistent — concurrent updates को tolerate करता है
Use Case	Single-threaded या externally synchronized	Multi-threaded environments where thread-safe operations चाहिए
Implementation Details	Uses array + linked lists/tree	Uses lock-stripping (Java 7) or CAS-based concurrency (Java 8+)

1 Thread Safety का फर्क

- **HashMap:** अगर multiple threads simultaneously modify करें, तो data corrupt हो सकता है (race condition), inconsistent data हो सकता है।
- **ConcurrentHashMap:** internally अलग-अलग segments या buckets पर fine-grained locking या lock-free techniques use करता है, जिससे multiple threads safe तरीके से read/write कर सकते हैं।

2 Null key और Null value

- HashMap में 1 null key और कई null values allowed हैं।
- ConcurrentHashMap में **null keys और null values allowed नहीं हैं।** (क्योंकि ये ambiguity create करता है कि null का मतलब "value not found" है या null actual value है।)

3 Performance

- Single-threaded में HashMap ज्यादा fast होता है क्योंकि कोई synchronization नहीं होता।
- Multi-threaded में ConcurrentHashMap बेहतर होता है क्योंकि ये concurrent access को efficiently handle करता है।

4 Iteration

- HashMap iterator **fail-fast** होता है, मतलब अगर एक thread iteration के दौरान map को modify करे तो ConcurrentModificationException throw करता है।
- ConcurrentHashMap का iterator **weakly consistent** होता है, जो concurrent modification tolerate करता है, और latest updates दिखा सकता है।

5 Java 7 vs Java 8 में Difference

- Java 7 में ConcurrentHashMap internally data को 16 segments में बाँटता था, हर segment पे अलग lock होता था। इसलिए multi-threading में contention कम होती थी।
- Java 8 में यह segments concept हटाकर पूरा map एक array में रखता है और **CAS (Compare-And-Swap)** और **synchronized** blocks का intelligent मिश्रण इस्तेमाल करता है ताकि performance improve हो।

6 Example

```
// HashMap (not thread-safe)
Map<String, String> map = new HashMap<>();
map.put("key", "value");

// ConcurrentHashMap (thread-safe)
Map<String, String> cmap = new ConcurrentHashMap<>();
cmap.put("key", "value");
```

Summary:

Aspect	HashMap	ConcurrentHashMap
Thread Safe?	No	Yes
Null keys allowed?	Yes (one)	No
Null values allowed?	Yes	No
Use Case	Single-threaded	Multi-threaded environment

अर्थात:

- अगर आपका environment single-threaded है या आप खुद synchronization handle कर रहे हो → **HashMap** use करो।
- अगर multi-threaded environment में बिना external synchronization के safe access चाहिए → **ConcurrentHashMap** best choice है।

ConcurrentHashMap vs Collections.synchronizedMap()

Aspect	ConcurrentHashMap	Collections.synchronizedMap()
Thread Safety	हाँ, internally thread-safe with fine-grained concurrency	हाँ, पूरा Map के ऊपर एक global lock लगाकर thread-safe बनाता है
Locking Mechanism	Bucket-level या segment-level locking / CAS (Java 8+)	Synchronized block (single lock) on entire Map
Performance	ज्यादा efficient, concurrent reads और writes allow करता है	कम efficient, क्योंकि एक बार में सिर्फ एक thread access कर सकता है
Null keys & values	नहीं allowed	null keys और null values allowed (अगर underlying Map allow करता है)
Iteration behavior	Weakly consistent iterator, concurrent modification tolerate करता है	Fail-fast iterator, concurrent modification पर ConcurrentModificationException आता है
Use case	Multi-threaded apps जहाँ high concurrency चाहिए	Simple thread-safe wrapper चाहिए, low concurrency वाले cases में ठीक
Internal Structure	Fine-grained locking, advanced concurrency techniques	Synchronized wrapper, internally synchronized method calls

Detailed Explanation:

1 Locking

- **ConcurrentHashMap** multiple locks लगाता है ताकि कई threads एक साथ अलग-अलग buckets या segments को update कर सकें।
- **SynchronizedMap** पूरा Map एक ही global lock के नीचे होता है, मतलब एक समय में केवल एक thread ही access कर सकता है।

2 Performance

- **ConcurrentHashMap** ज्यादा scalable और faster होता है high concurrency वाले environment में।
- **SynchronizedMap** slow हो सकता है क्योंकि सारे operations synchronized होते हैं, और contention बढ़ जाता है।

3 Null Keys और Values

- **ConcurrentHashMap** null keys और values allow नहीं करता।

- **SynchronizedMap** underlying Map के behavior को follow करता है, इसलिए HashMap है तो null allowed है।
-

4 Iteration Behavior

- **ConcurrentHashMap** का iterator weakly consistent होता है, जो concurrent modification tolerate करता है और partial updated data भी दिखा सकता है।
 - **SynchronizedMap** का iterator fail-fast होता है, concurrent modification पर `ConcurrentModificationException` देगा।
-

5 Usage Example

```
// SynchronizedMap
Map<String, String> syncMap = Collections.synchronizedMap(new HashMap<>());
synchronized(syncMap) {
    for (String key : syncMap.keySet()) {
        System.out.println(key + " = " + syncMap.get(key));
    }
}

// ConcurrentHashMap
Map<String, String> concurrentMap = new ConcurrentHashMap<>();
for (String key : concurrentMap.keySet()) {
    System.out.println(key + " = " + concurrentMap.get(key));
}
```

ध्यान दें: SynchronizedMap के साथ iteration के दौरान manual synchronization चाहिए।

Summary:

Feature	ConcurrentHashMap	Collections.synchronizedMap()
Thread-safety model	Fine-grained, concurrent access	Coarse-grained, single lock
Performance	High concurrency possible	Lower performance under contention
Null keys/values	Not allowed	Allowed if underlying map allows
Iteration	Weakly consistent, safe	Fail-fast, needs manual sync
Use case	High concurrency apps	Simple synchronization need

मतलब:

अगर आपको high concurrency के साथ efficient, thread-safe map चाहिए → **ConcurrentHashMap** best है।

अगर बस simple thread-safe wrapper चाहिए और concurrency कम है → **Collections.synchronizedMap()** काम कर सकता है।

1 Hashtable vs HashMap

Feature	Hashtable	HashMap
Thread Safety	Thread-safe (synchronized methods)	Not thread-safe (unless externally synchronized)
Null keys & values	Not allowed	Allows 1 null key & multiple null values
Performance	Slower due to synchronization	Faster, no synchronization by default
Iterator behavior	Enumerator (legacy, fail-fast no)	Fail-fast iterator
Legacy	Yes, part of legacy java.util	Part of Java Collections Framework (JDK 1.2+)

2 Hashtable vs ConcurrentHashMap

Feature	Hashtable	ConcurrentHashMap
Thread Safety	Synchronized on whole object	Fine-grained locking / CAS based
Performance	Slower (coarse lock)	Faster, supports high concurrency
Null keys & values	Not allowed	Not allowed
Iterator behavior	Enumerator	Weakly consistent iterator
Use case	Legacy, rarely used now	Modern concurrent access

3 Hashtable vs Collections.synchronizedMap(new HashMap<>())

Feature	Hashtable	SynchronizedMap wrapper
Thread Safety	Internal synchronization (methods)	External synchronization (wrapper)
Null keys & values	Not allowed	Depends on underlying Map (usually allowed for HashMap)
Performance	Slower, single lock	Similar, but allows external synchronization flexibility
Legacy	Legacy class	Wrapper introduced post Collections Framework

Summary Table (Hashtable vs Others)

Feature	Hashtable	HashMap	ConcurrentHashMap	SynchronizedMap
Thread-safe?	Yes (synchronized)	No	Yes (fine-grained)	Yes (coarse lock)
Null keys allowed?	No	Yes (one key)	No	Depends underlying map
Performance	Slow (coarse locking)	Fast	Fast (high concurrency)	Slow (single lock)
Iterator	Enumerator	Fail-fast iterator	Weakly consistent	Fail-fast iterator
Use case	Legacy	General purpose	High concurrency apps	Simple synchronization

When to use Hashtable?

- **Mostly legacy code में:** क्योंकि आजकल HashMap + ConcurrentHashMap ज्यादा popular हैं।
- अगर आपको बहुत basic thread-safe map चाहिए और legacy compatibility चाहिए।
- Otherwise, **ConcurrentHashMap** या **synchronizedMap** ज्यादा बेहतर विकल्प हैं।

Hashtable: Internal Working और Synchronization Mechanism

1) Hashtable क्या है?

- Hashtable Java का एक legacy class है जो key-value pairs को store करता है।
- ये thread-safe है क्योंकि इसके सारे public methods synchronized होते हैं।
- Hashtable internally एक array of buckets (similar to HashMap) use करता है।

2) Data Structure

- Hashtable में भी एक **array** होता है, जिसमें हर index पर **Entry** objects chained होते हैं (linked list)।
 - हर Entry में key, value, hash code, और next reference होती है।
-

3 Synchronization Mechanism

Synchronized Methods

- Hashtable के सारे public methods जैसे `get()`, `put()`, `remove()` आदि **synchronized** होते हैं।
- इसका मतलब ये है कि एक समय में सिर्फ एक thread ही Hashtable के किसी भी method को access कर सकता है।

```
public synchronized V put(K key, V value) {  
    // implementation  
}
```

Effect

- ये पूरी Hashtable को lock कर देता है जब भी कोई thread कोई method call करता है।
 - इसलिए, multi-threaded environment में Hashtable thread-safe है।
 - लेकिन यह **coarse-grained locking** है, यानी पूरा object lock होता है, जिससे concurrency कम होती है।
-

4 put() Operation (Internals)

- जब आप `put(key, value)` call करते हैं:
 - Hashtable का lock acquire होता है (method synchronized है)।
 - Key का hashCode निकाला जाता है।
 - Bucket index calculate होता है।
 - अगर उस bucket में पहले से entry है, तो linked list में iterate कर key के बराबर वाले node की खोज होती है।
 - अगर key मिलती है तो उसका value update होता है।
 - नहीं तो नई Entry list में add होती है।
 - फिर lock release होता है।
-

5 get() Operation (Internals)

- जब आप `get(key)` call करते हैं:
 - Hashtable lock acquire करता है।
 - Key का hashCode निकालकर bucket index पता चलता है।
 - उस bucket में linked list traverse करके key को खोजा जाता है।
 - Value return की जाती है।
 - Lock release किया जाता है।
-

6 Resizing Mechanism

- जब Hashtable का load factor exceed हो जाता है, तो Hashtable अपनी capacity double करता है।
 - ये operation भी synchronized होता है ताकि resize के दौरान कोई thread interfere न करे।
 - Resizing में existing entries को नए bigger array में rehash किया जाता है।
-

7 क्या होता है Coarse-Grained Locking?

- Hashtable में पूरे object पर lock लगता है।
 - मतलब एक thread `put()` कर रहा हो तो कोई दूसरा thread `get()` या `put()` नहीं कर सकता।
 - इससे performance degrade हो सकती है, खासकर जब कई threads एक साथ access कर रहे हों।
-

8 Hashtable का drawback

- Thread-safety तो मिलती है लेकिन concurrency कम हो जाती है।
 - Modern concurrent collections (जैसे `ConcurrentHashMap`) बेहतर तरीके से concurrency handle करते हैं।
-

Visual Summary

```
Thread 1: calls put() --> acquires Hashtable lock --> modifies map --> releases lock
Thread 2: calls get() --> waits till Thread 1 releases lock --> acquires lock --> reads data --> releases lock
```

Conclusion:

- Hashtable**: synchronized methods के कारण thread-safe, लेकिन coarse-grained locking से slow under high concurrency।
- Modern Java में **ConcurrentHashMap** बेहतर alternative है क्योंकि वो fine-grained locking और lock-free techniques use करता है।

`equals()` और `hashCode()` क्या हैं और क्यों ज़रूरी हैं?

1 equals() method

- ये method दो objects को **logical equality** से compare करता है।
- मतलब, यह चेक करता है कि दो objects "समान" हैं या नहीं, चाहे वे memory में अलग-अलग हों।
- Default behavior (जो Object class से inherited होता है): यह केवल reference equality (address comparison) करता है।
- जब हम चाहते हैं कि दो objects का content समान हो, तो equals() को override करते हैं।

Example:

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

    @Override
    public boolean equals(Object o) {
        if (this == o) return true; // same reference
        if (o == null || getClass() != o.getClass()) return false;
        Person person = (Person) o;
        return age == person.age && name.equals(person.name);
    }
}
```

2 hashCode() method

- यह method object के लिए एक integer value (hash code) generate करता है।
- यह value object की internal state के आधार पर unique होनी चाहिए (as much as possible)।
- Java Collections Framework में (जैसे HashMap), objects को hash code के basis पर buckets में डालते हैं।

Important:

- अगर दो objects equals() method से बराबर हैं, तो उनका hashCode() भी **same** होना चाहिए।
 - लेकिन अगर hashCode same है, तो जरूरी नहीं कि वे equals भी हों (collision हो सकता है)।
-

3 equals() और hashCode() के बीच Contract (Rule)

Java के specification के मुताबिक:

- **Consistency:** अगर दो objects बराबर हैं (equals true), तो उनका hashCode भी बराबर होना चाहिए।

- **Non-equality:** अगर दो objects बराबर नहीं हैं, तो उनके `hashCode` अलग हो सकते हैं (ideal) लेकिन जरूरी नहीं।
 - **Consistency over time:** एक object का `hashCode` execution के दौरान बार-बार एक जैसा रहना चाहिए।
-

4 क्यों जरूरी हैं ये दो methods?

Hash-based Collections में काम कैसे करते हैं?

- **HashMap/HashSet** में किसी object को add करते समय:
 1. Object का `hashCode()` निकाला जाता है ताकि bucket पता चले।
 2. उस bucket में जाकर `equals()` से चेक करते हैं कि वही object पहले से मौजूद है या नहीं।
 - अगर आपने सिर्फ `equals()` override किया और `hashCode()` नहीं किया, तो HashMap में duplicate entries हो सकती हैं क्योंकि अलग-अलग `hashCodes` के कारण अलग buckets में डाल देता है।
-

5 एक Simple Example:

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

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof Employee)) return false;
        Employee emp = (Employee) o;
        return this.id == emp.id && this.name.equals(emp.name);
    }

    @Override
    public int hashCode() {
        return Objects.hash(id, name);
    }
}
```

- यहाँ `hashCode()` और `equals()` दोनों override किए गए हैं।
 - इससे अगर आप HashSet में दो Employees डालेंगे जिनका id और name same है, तो duplicate नहीं होगा।
-

6 Summary Table

Aspect	<code>equals()</code>	<code>hashCode()</code>
Purpose	Logical equality check	Generate integer hash for object

Aspect	<code>equals()</code>	<code>hashCode()</code>
Returns	boolean	int
Used in	Checking equality between objects	Bucketing in hash-based collections
Contract	Reflexive, symmetric, transitive	Consistent with equals
Must Override together?	Yes (for hash-based collections)	Yes

Extra Tips:

- Override करते वक्त दोनों methods को साथ में implement करें।
- Use `Objects.equals()` और `Objects.hash()` helpers (Java 7+) से implementation आसान हो जाती है।
- Hash code generation में fields को ठीक से consider करें जिनसे object की equality define होती है।