**Spring Bean Scopes with Real-Time Examples**

In Spring, **scopes** define the lifecycle and visibility of a bean. There are **five major scopes** in Spring:

1. **Singleton** (Default)
2. **Prototype**
3. **Request** (Only for Web Applications)
4. **Session** (Only for Web Applications)
5. **Application** (Only for Web Applications)

**1. Singleton Scope (Default)**

**Definition**

* Only **one instance** of the bean is created for the entire Spring container.
* **Same instance** is shared across all requests.

**Real-Time Example: Logging Service**

A logging service should use the **same instance** throughout the application.

package com.example.service;

import lombok.extern.slf4j.Slf4j;

import org.springframework.stereotype.Service;

@Slf4j

@Service // Default scope is Singleton

public class LoggingService {

public LoggingService() {

log.info("LoggingService instance created");

}

public void logMessage(String message) {

log.info("Log: {}", message);

}

}

**Test Singleton Behavior**

@RestController

@RequestMapping("/api")

public class LogController {

@Autowired

private LoggingService loggingService1;

@Autowired

private LoggingService loggingService2;

@GetMapping("/log")

public String logTest() {

loggingService1.logMessage("Hello from loggingService1");

loggingService2.logMessage("Hello from loggingService2");

return loggingService1 == loggingService2 ? "Same instance" : "Different instances";

}

}

✅ **Output:** Same instance  
*(Only one object is created)*

**2. Prototype Scope**

**Definition**

* A **new instance** of the bean is created **every time** it is requested.

**Real-Time Example: Random Number Generator**

Each user should get a **new random number instance**.

package com.example.service;

import org.springframework.stereotype.Service;

import java.util.Random;

import org.springframework.context.annotation.Scope;

@Service

@Scope("prototype") // Creates a new instance every time

public class RandomNumberService {

private final int randomNumber;

public RandomNumberService() {

this.randomNumber = new Random().nextInt(100);

}

public int getRandomNumber() {

return randomNumber;

}

}

**Test Prototype Behavior**

@RestController

@RequestMapping("/api")

public class RandomController {

@Autowired

private RandomNumberService randomService1;

@Autowired

private RandomNumberService randomService2;

@GetMapping("/random")

public String randomTest() {

return "Random1: " + randomService1.getRandomNumber() +

", Random2: " + randomService2.getRandomNumber();

}

}

✅ **Output:**

Random1: 42, Random2: 17

*(Different instances are created)*

**3. Request Scope (Web Only)**

**Definition**

* A **new instance** of the bean is created **for each HTTP request**.

**Real-Time Example: Tracking Request ID**

Each request should have **a unique ID**.

package com.example.service;

import org.springframework.context.annotation.Scope;

import org.springframework.stereotype.Component;

import java.util.UUID;

@Component

@Scope("request") // Creates a new instance per request

public class RequestTracker {

private final String requestId;

public RequestTracker() {

this.requestId = UUID.randomUUID().toString();

}

public String getRequestId() {

return requestId;

}

}

**Test Request Scope**

@RestController

@RequestMapping("/api")

public class RequestController {

@Autowired

private RequestTracker requestTracker;

@GetMapping("/request")

public String requestTest() {

return "Request ID: " + requestTracker.getRequestId();

}

}

✅ **Test with Multiple Requests**

GET /api/request → Request ID: abc-123

GET /api/request → Request ID: xyz-456

*(Each request has a different ID)*

**4. Session Scope (Web Only)**

**Definition**

* A **new instance** of the bean is created **per user session**.

**Real-Time Example: Shopping Cart**

Each user should have **a separate cart**.

package com.example.service;

import org.springframework.context.annotation.Scope;

import org.springframework.stereotype.Component;

import java.util.ArrayList;

import java.util.List;

@Component

@Scope("session") // Creates a new instance per session

public class ShoppingCartService {

private final List<String> items = new ArrayList<>();

public void addItem(String item) {

items.add(item);

}

public List<String> getItems() {

return items;

}}

**Test Session Scope**

@RestController

@RequestMapping("/api/cart")

public class CartController {

@Autowired

private ShoppingCartService shoppingCart;

@GetMapping("/add/{item}")

public String addItem(@PathVariable String item) {

shoppingCart.addItem(item);

return "Item added: " + item;

}

@GetMapping("/items")

public List<String> getCartItems() {

return shoppingCart.getItems();

}}

✅ **Test with Same User**

GET /api/cart/add/apple → "Item added: apple"

GET /api/cart/add/mango → "Item added: mango"

GET /api/cart/items → ["apple", "mango"]

*(Same session retains items)*

✅ **Test with Different Users**

User 1: GET /api/cart/items → ["apple", "mango"]

User 2: GET /api/cart/items → []

*(Each session has a different cart)*

**5. Application Scope (Web Only)**

**Definition**

* A **single instance** is shared **across all sessions** and **requests**.

**Real-Time Example: Site Visit Counter**

The **same counter** should be shared across users.

package com.example.service;

import org.springframework.context.annotation.Scope;

import org.springframework.stereotype.Component;

@Component

@Scope("application") // Shared across all users

public class VisitCounterService {

private int counter = 0;

public void increment() {

counter++;

}

public int getCount() {

return counter;

}

}

**Test Application Scope**

@RestController

@RequestMapping("/api/visit")

public class VisitController {

@Autowired

private VisitCounterService visitCounter;

@GetMapping("/increment")

public String incrementVisit() {

visitCounter.increment();

return "Total Visits: " + visitCounter.getCount();

}

}

✅ **Test Across Users**

User 1: GET /api/visit/increment → Total Visits: 1

User 2: GET /api/visit/increment → Total Visits: 2

User 3: GET /api/visit/increment → Total Visits: 3

*(Shared counter across all users)*

**Summary of Scopes**

| **Scope** | **Instance Created** | **Lifecycle** | **Real-Time Example** |
| --- | --- | --- | --- |
| **Singleton** | **One** per container | App startup to shutdown | Logging Service |
| **Prototype** | **New** per request | Created & destroyed immediately | Random Number Generator |
| **Request** | **New** per HTTP request | Destroyed after request | Request Tracker |
| **Session** | **New** per user session | Destroyed after session | Shopping Cart |
| **Application** | **One** per app | App startup to shutdown | Site Visit Counter |

**Final Thoughts**

* Use **Singleton** for shared services (e.g., Logging).
* Use **Prototype** for frequently changing data (e.g., Random Numbers).
* Use **Request** for per-request data (e.g., Request ID).
* Use **Session** for user-specific data (e.g., Shopping Cart).
* Use **Application** for shared app-wide data (e.g., Visitor Count).

### ****❓ Why Can't We Use**** @Repository ****Instead of**** @Service ****and Vice Versa?****

Technically, **Spring does not enforce restrictions** on using @Repository or @Service. However, **each annotation serves a specific purpose**, and misusing them **affects maintainability, readability, and behavior**.

## ****1️⃣**** @Repository ****vs.**** @Service ****- Purpose and Differences****

| **Annotation** | **Purpose** | **Special Features** |
| --- | --- | --- |
| **@Repository** | Used for **DAO (Data Access Object) Layer** to interact with the database | **Automatic translation of JDBC exceptions** into Spring DataAccessException |
| **@Service** | Used for **Business Logic Layer** where business rules and operations are implemented | No extra behavior, just for readability and separation of concerns |

## ****2️⃣ Why Not Use**** @Repository ****Instead of**** @Service****?****

🚫 **Problem:** @Repository is meant for **data access**, not business logic.

✅ **Example (Correct Use of @Repository and @Service)**

@Repository

public class UserRepository {

public User findById(Long id) {

// Fetch user from DB (Pretend logic)

return new User(id, "Devendra");

}

}

@Service

public class UserService {

private final UserRepository userRepository;

public UserService(UserRepository userRepository) {

this.userRepository = userRepository;

}

public String getUserGreeting(Long id) {

User user = userRepository.findById(id);

return "Hello, " + user.getName();

}

}

✅ **Correct separation**: UserRepository handles **DB operations**, UserService contains **business logic**.

🚫 **If We Use @Repository on UserService**

@Repository // ❌ Wrong annotation, because this is not a DAO class

public class UserService { ... }

🔴 **Issues:**

* Misleads developers because **it is not a DAO**.
* Exception handling behavior of @Repository (JDBC exception translation) **is unnecessary** in business logic.

## ****3️⃣ Why Not Use**** @Service ****Instead of**** @Repository****?****

🚫 **Problem:** @Service **does not provide exception translation** for DAO operations.

✅ **Example (Correct Use of @Repository)**

@Repository

public class ProductRepository {

public Product findById(Long id) {

// Code to fetch product from DB

return new Product(id, "Laptop");

}

}

🚫 **If We Use @Service on ProductRepository**

@Service // ❌ Wrong annotation, because this is a DAO

public class ProductRepository { ... }

🔴 **Issues:**

* **No automatic JDBC exception translation** (DataAccessException handling is lost).
* Code structure becomes unclear.

## ****4️⃣ Key Takeaways****

✅ @Repository → **Use for DAO layer** (Exception Translation)  
✅ @Service → **Use for Business Logic layer** (No Extra Features)  
🚫 **Do not interchange them**, as it reduces code clarity and expected behavior.

**POJO (Plain Old Java Object) in Spring**

**1. What is a POJO?**

POJO stands for **Plain Old Java Object**, which is a simple Java class that follows certain basic principles:

* It should not extend specific classes (like HttpServlet or EJB classes).
* It should not implement special interfaces (except standard Java interfaces like Serializable).
* It should have private fields with public getter and setter methods.
* It may have constructors, methods, and annotations but should remain a simple data container.

**2. Why Use POJOs in Spring?**

Spring heavily relies on POJOs because:

* They make the application more **loosely coupled** and **maintainable**.
* They allow **dependency injection (DI)** for better **testability**.
* They help in **encapsulation** of data.

**3. POJO in Different Layers of a Spring Application**

Spring uses POJOs in multiple layers of an application:

**a. Model Layer (Entity Class)**

POJOs are often used as entity classes in a Spring application, representing database tables.

public class Employee {

private int id;

private String name;

private String department;

public Employee() {}

public Employee(int id, String name, String department) {

this.id = id;

this.name = name;

this.department = department;

}

public int getId() { return id; }

public void setId(int id) { this.id = id; }

public String getName() { return name; }

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

public String getDepartment() { return department; }

public void setDepartment(String department) { this.department = department; }

}

**b. Service Layer (Business Logic)**

POJOs can be used in the service layer to encapsulate business logic.

Example:

@Service

public class EmployeeService {

public String getEmployeeDetails(Employee emp) {

return "Employee: " + emp.getName() + ", Department: " + emp.getDepartment();

}

}

**c. Controller Layer (Handling HTTP Requests)**

Spring controllers use POJOs to map JSON data from client requests.

Example:

@RestController

@RequestMapping("/employee")

public class EmployeeController {

@PostMapping("/create")

public String createEmployee(@RequestBody Employee emp) {

return "Employee " + emp.getName() + " created successfully!";

}

}

### 4. POJO vs JavaBean vs DTO

| **Feature** | **POJO (Plain Old Java Object)** | **JavaBean** | **DTO (Data Transfer Object)** |
| --- | --- | --- | --- |
| **Definition** | A simple Java object without specific restrictions. | A POJO with additional rules (serialization, no-arg constructor, getters/setters). | A specialized POJO used for transferring data between layers. |
| **Encapsulation** | Optional | Required | Required |
| **Fields** | Can be public or private | Must be private | Must be private |
| **Getters/Setters** | Optional | Mandatory | Mandatory |
| **Constructor** | Not required | Must have a no-arg constructor | Not required but recommended |
| **Implements Serializable** | Optional | Recommended (for JavaBeans standard) | Often implemented for data transfer |
| **Annotations** | Not required | Not required but used in frameworks | Often annotated (@Data, @Getter, etc.) |
| **Usage** | Any simple object in Java | Used in frameworks like JSP, JSF | Transfers data between layers (e.g., API, DB) |
| **Example Use Case** | General-purpose Java objects | Java EE, Spring Beans | REST APIs, microservices, database communication |

**5. POJO with Spring Boot**

Spring Boot automatically maps JSON requests to POJOs using Jackson.

Example of POJO with Lombok (to avoid boilerplate code):

import lombok.AllArgsConstructor;

import lombok.Data;

import lombok.NoArgsConstructor;

@Data

@NoArgsConstructor

@AllArgsConstructor

public class Product {

private int id;

private String name;

private double price;

}

* @Data: Generates getters, setters, toString(), and equals() methods.
* @NoArgsConstructor: Generates a no-argument constructor.
* @AllArgsConstructor: Generates a constructor with all fields.

**6. Conclusion**

* POJOs are simple Java objects without complex dependencies.
* Spring uses POJOs in model, service, and controller layers.
* POJOs help in making applications **modular, testable, and maintainable**.
* Spring Boot and Lombok simplify working with POJOs.

**Spring Boot 3 and Spring 6 Upgrade Notes**

### ****Upgrade Requirements:****

1. **Upgrade to Java 17 or Higher**
   * Spring Boot 3 and Spring 6 require a minimum of **Java 17**.
2. **Update Package Imports to Jakarta EE Namespace**
   * Spring Boot 3 and Spring 6 use Jakarta EE instead of Java EE.
   * **Hibernate/JPA Code:**
     + Replace: import javax.persistence.\*;
     + With: import jakarta.persistence.\*;
   * **@PostConstruct and @PreDestroy:**
     + Replace:  
       import javax.annotation.PostConstruct;  
       import javax.annotation.PreDestroy;
     + With:  
       import jakarta.annotation.PostConstruct;  
       import jakarta.annotation.PreDestroy;
   * **Hibernate Validator:**
     + Replace: import javax.validation.constraints.\*;
     + With: import jakarta.validation.constraints.\*;
     + Use **Hibernate Validator version 8.x**.
3. **Upgrade to Tomcat 10 (For Spring MVC Apps - Non Boot)**
   * Spring MVC applications require **Tomcat 10** due to Jakarta EE support.
   * Download Tomcat 10 from: [Apache Tomcat 10](https://tomcat.apache.org/download-10.cgi)

By following these updates, your code will be compatible with **Spring Boot 3** and **Spring 6**.

**How IoC (Inversion of Control) Works in the Background in Spring**

In Spring, **IoC (Inversion of Control)** is a design principle where the **control of object creation and dependency management is transferred** from the developer to the **Spring Container**. This mechanism allows for better **loose coupling** and **dependency injection**.

**1. IoC in Action - Behind the Scenes**

When you create a Spring application, the following steps happen in the background:

**Step 1: Spring Container Initialization**

* When the application starts, Spring **creates an IoC container** (like ApplicationContext).
* This container is responsible for managing **beans** (Java objects).
* Spring **scans and registers beans** defined using **XML configuration, Java-based configuration, or annotations**.

Example:

java

ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);

Here, context is the IoC container managing beans.

**Step 2: Bean Creation & Management**

* The IoC container **automatically detects beans** (e.g., classes with @Component, @Service, etc.).
* Spring **instantiates the beans** and manages their lifecycle.
* It stores the beans in the **Bean Factory (Singleton or Prototype scope).**

Example:

java

@Component

public class EmployeeService {

public void serve() {

System.out.println("Serving Employee...");

}

}

Spring detects this class and **creates an instance automatically**.

**Step 3: Dependency Injection (DI)**

* When a class requires another bean, Spring **injects the dependency automatically**.
* The container **looks up dependencies** and **resolves them**.

Example:

java

@Service

public class CompanyService {

private final EmployeeService employeeService;

@Autowired

public CompanyService(EmployeeService employeeService) {

this.employeeService = employeeService;

}

public void process() {

employeeService.serve();

}

}

* @Autowired tells Spring to **inject the EmployeeService bean automatically**.

**Step 4: Bean Lifecycle Management**

* Spring **manages the lifecycle** of beans, including:
  + **Initialization (@PostConstruct)**
  + **Usage**
  + **Destruction (@PreDestroy)**

Example:

@Component

public class DatabaseConnection {

@PostConstruct

public void init() {

System.out.println("Initializing DB connection...");

}

@PreDestroy

public void destroy() {

System.out.println("Closing DB connection...");

}

}

Spring automatically calls init() after bean creation and destroy() before shutting down.

**2. Types of IoC Containers in Spring**

Spring provides two types of IoC containers:

1. **BeanFactory** (Lightweight, used for simple applications) BeanFactory factory = new ClassPathXmlApplicationContext("beans.xml");
2. **ApplicationContext** (More advanced, supports AOP, event propagation) ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);

**3. IoC vs Traditional Object Creation**

| **Feature** | **Traditional Java** | **Spring IoC** |
| --- | --- | --- |
| Object Creation | Manually using new keyword | Automatically managed by Spring |
| Dependency Management | Developer handles dependencies | Spring injects dependencies |
| Lifecycle Control | Developer must manage manually | Spring manages lifecycle |
| Flexibility | Hardcoded dependencies | Loose coupling with DI |

Example (Without IoC):

public class Main {

public static void main(String[] args) {

EmployeeService employeeService = new EmployeeService(); // Manual object creation

CompanyService companyService = new CompanyService(employeeService);

}

}

With IoC:

ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);

CompanyService companyService = context.getBean(CompanyService.class);

companyService.process(); // IoC manages dependencies

**Conclusion**

* **IoC allows Spring to control object creation & dependency injection**.
* **Spring Container manages bean lifecycle & dependencies automatically**.
* **It leads to more modular, testable, and maintainable applications**

**Code to Interface in Spring - Detailed Explanation**

**1. What is "Code to Interface" in Spring?**

**Code to Interface** is a design principle where we write code using **interfaces** instead of directly referring to concrete classes. This improves **loose coupling**, making the code more flexible, maintainable, and testable.

**Spring Framework promotes this by using Dependency Injection (DI) to inject dependencies via interfaces rather than concrete implementations.**

**2. Why Use Code to Interface?**

✅ **Loose Coupling** → The implementation can be changed easily.  
✅ **Easier Unit Testing** → You can mock dependencies using interfaces.  
✅ **Better Maintainability** → Changing an implementation does not affect dependent classes.  
✅ **Follows SOLID Principles** → Especially the **Dependency Inversion Principle (DIP)**.

**3. Example of Code to Interface in Spring**

Let's implement **Code to Interface** using Spring IoC and Dependency Injection.

**Step 1: Create an Interface**

We define a PaymentService interface that provides a method for processing payments.

java

public interface PaymentService {

void processPayment(double amount);

}

**Step 2: Implement the Interface (Concrete Classes)**

We create two implementations: **CreditCardPaymentService** and **PayPalPaymentService**.

java

import org.springframework.stereotype.Service;

@Service

public class CreditCardPaymentService implements PaymentService {

@Override

public void processPayment(double amount) {

System.out.println("Processing credit card payment of $" + amount);

}

}

java

import org.springframework.stereotype.Service;

@Service

public class PayPalPaymentService implements PaymentService {

@Override

public void processPayment(double amount) {

System.out.println("Processing PayPal payment of $" + amount);

}

}

**Step 3: Inject the Dependency in a Client Class**

Instead of directly creating objects, we inject the PaymentService **interface** using @Autowired.

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

@Service

public class ShoppingCartService {

private final PaymentService paymentService;

@Autowired // Spring will automatically inject an implementation of PaymentService

public ShoppingCartService(PaymentService paymentService) {

this.paymentService = paymentService;

}

public void checkout(double amount) {

System.out.println("Checking out with amount: $" + amount);

paymentService.processPayment(amount); // Calls the method dynamically

}

}

**Step 4: Configure and Run the Application**

We create a Spring Boot application to demonstrate how IoC injects dependencies.

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.context.ApplicationContext;

@SpringBootApplication

public class PaymentApplication {

public static void main(String[] args) {

ApplicationContext context = SpringApplication.run(PaymentApplication.class, args);

ShoppingCartService shoppingCartService = context.getBean(ShoppingCartService.class);

shoppingCartService.checkout(100.0); // Will process payment dynamically

}

}

**4. How Spring Handles Dependency Injection?**

When Spring starts:

1. It **scans** the classes annotated with @Service, @Component, etc.
2. It finds that CreditCardPaymentService and PayPalPaymentService implement PaymentService.
3. It **injects one implementation automatically** into ShoppingCartService (default is the first found).
4. The checkout() method calls processPayment() **without knowing the concrete class**.

**5. What if Multiple Implementations Exist?**

If we have multiple implementations (CreditCard & PayPal), Spring doesn't know which one to inject. To solve this:

**Option 1: Use @Qualifier**

Specify the exact implementation:

@Autowired

@Qualifier("creditCardPaymentService")

private PaymentService paymentService;

**Option 2: Use @Primary**

Mark one implementation as the default:

@Service

@Primary

public class CreditCardPaymentService implements PaymentService {

// Implementation...

}

**6. Key Takeaways**

* **Code to Interface promotes loose coupling** by depending on interfaces instead of concrete classes.
* **Spring IoC container handles dependency injection**, automatically injecting the required implementation.
* **Multiple implementations can be managed** using @Qualifier or @Primary.
* **It makes unit testing easier** since we can inject mock implementations.

**7. Example Unit Test**

Since we're coding to an interface, we can easily test ShoppingCartService using a mock implementation.

java

import static org.mockito.Mockito.\*;

import org.junit.jupiter.api.Test;

public class ShoppingCartServiceTest {

@Test

void testCheckout() {

PaymentService mockPaymentService = mock(PaymentService.class);

ShoppingCartService shoppingCart = new ShoppingCartService(mockPaymentService);

shoppingCart.checkout(200.0);

verify(mockPaymentService).processPayment(200.0); // Verify method was called

}

}

Here, we **mock** PaymentService instead of relying on a real implementation.

**8. Summary**

| **Feature** | **Without Interface** | **With Interface (Best Practice)** |
| --- | --- | --- |
| Coupling | Tightly coupled | Loosely coupled |
| Testability | Hard to test | Easy to mock & test |
| Flexibility | Hard to switch implementations | Easy to swap implementations |
| Maintainability | High dependency | Easier to maintain |

**Coding to an interface + Spring IoC = Highly scalable & maintainable applications!** 🚀

**How the Code Works in the Background (Spring IoC & Dependency Injection)**

Your Spring application follows the **Inversion of Control (IoC) and Dependency Injection (DI)** principles. Let's go step by step to understand **how it works in the background.** 🚀

**1. Application Starts: Spring Context is Created**

**Code Execution:**

ClassPathXmlApplicationContext context =

new ClassPathXmlApplicationContext("applicationContext.xml");

**Behind the Scenes:**

1. Spring **reads the applicationContext.xml file**.
2. It **creates and manages all beans** (objects) inside the **IoC container**.
3. It **initializes** the beans based on the configuration.

**2. Bean Creation & Dependency Injection**

**Beans Defined in applicationContext.xml:**

<bean id="myFortuneService"

class="spring\_01\_demo.springdemo.HappyFortuneService">

</bean>

<bean id="myCoach" class="spring\_01\_demo.springdemo.TrackCoach">

<constructor-arg ref="myFortuneService" />

</bean>

**What Happens Internally?**

1. Spring **creates an instance** of HappyFortuneService:

HappyFortuneService myFortuneService = new HappyFortuneService();

1. Then, Spring **creates an instance** of TrackCoach, passing the myFortuneService object via constructor injection:

TrackCoach myCoach = new TrackCoach(myFortuneService);

**3. Bean Retrieval from Spring Container**

**Code Execution:**

Coach theCoach = context.getBean("myCoach", Coach.class);

**Behind the Scenes:**

* Spring **retrieves the bean** from the IoC container.
* Since TrackCoach implements Coach, Spring returns the object as type Coach.

Internally, it's equivalent to:

Coach theCoach = (Coach) springContainer.getBean("myCoach");

**4. Method Calls & Dependency Usage**

**Code Execution:**

System.out.println(theCoach.getDailyWorkout());

System.out.println(theCoach.getDailyFortune());

**Behind the Scenes:**

1. When calling theCoach.getDailyWorkout(), the method from TrackCoach is executed:

java

public String getDailyWorkout() {

return "Run a hard 5k";

}

Output:Run a hard 5k

1. When calling theCoach.getDailyFortune(), it calls getFortune() from the injected HappyFortuneService:

public String getDailyFortune() {

return "Just Do It: " + fortuneService.getFortune();

}

Since HappyFortuneService.getFortune() returns "Today is your lucky day!", the final output is:

vbnet

Just Do It: Today is your lucky day!

**5. Context Shutdown**

**Code Execution:**

context.close();

**Behind the Scenes:**

1. Spring **calls the destroy lifecycle methods** (if any).
2. The **Spring container is shut down**, releasing resources.

**Summary of What Happens in the Background**

| **Step** | **What Happens in Background** |
| --- | --- |
| **1. Load Context** | Spring reads applicationContext.xml, sets up the IoC container. |
| **2. Create Beans** | Spring creates HappyFortuneService and TrackCoach. |
| **3. Inject Dependencies** | TrackCoach gets HappyFortuneService injected via constructor. |
| **4. Retrieve Bean** | Spring returns TrackCoach when calling context.getBean("myCoach"). |
| **5. Execute Methods** | Calls getDailyWorkout() & getDailyFortune(), using the injected dependency. |
| **6. Close Context** | Spring releases resources and shuts down. |

**Why is This Important?**

✅ **Loose Coupling** → You can change HappyFortuneService without modifying TrackCoach.  
✅ **Easier Maintenance** → Configuration is externalized in XML (or Java-based configuration).  
✅ **Scalability** → You can replace TrackCoach with another implementation (e.g., BaseballCoach) without changing the main code.

----------------------------------------------------------------------------------------------------------------------------------------------------------------

<bean id="myFortuneService"

class="com.luv2code.springdemo.HappyFortuneService">

</bean>

<bean id="myCoach"

class="com.luv2code.springdemo.BaseballCoach">

<constructor-arg ref="myFortuneService" />

</bean>

HappyFortuneService myFortuneService = new HappyFortuneService();

BaseballCoach myCoach = new BaseballCoach(myFortuneService);

<bean id="myFortuneService"

class="com.luv2code.springdemo.HappyFortuneService">

</bean>

<bean id="myCricketCoach"

class="com.luv2code.springdemo.CricketCoach">

<property name="fortuneService" ref="myFortuneService" />

</bean>

HappyFortuneService myFortuneService = new HappyFortuneService();

CricketCoach myCricketCoach = new CricketCoach();

myCricketCoach.setFortuneService(myFortuneService);

The **no-argument constructor** in Spring-managed beans serves several purposes:

**1. Required for JavaBean Specification**

* Spring often follows the **JavaBean convention**, which recommends a **no-arg constructor** so that objects can be instantiated easily and populated via setters or dependency injection.

**2. Allows Spring to Create Bean Instances**

* When using **Spring IoC (Inversion of Control)**, Spring **first creates an object using the no-arg constructor**, then injects dependencies (if needed).
* If a **constructor with arguments is present**, but there is **no no-arg constructor**, you **must define explicit constructor-based injection** in Spring.

**3. Default Constructor is Needed for Some Proxying Mechanisms**

* Spring uses **CGLIB proxying** for beans that are **not interfaces** (e.g., when using @Transactional).
* CGLIB requires a **no-arg constructor** to generate a subclass proxy.

**4. Helpful for Serialization & Reflection**

* Some frameworks (e.g., Hibernate, Jackson, JPA) require a no-arg constructor for creating instances via reflection or deserialization.

**Example Scenario**

**Class with No-Arg Constructor (Works Fine with Spring)**

public class TrackCoach implements Coach {

private FortuneService fortuneService;

public TrackCoach() {

// No-arg constructor (needed for Spring)

}

public TrackCoach(FortuneService fortuneService) {

this.fortuneService = fortuneService;

}

@Override

public String getDailyWorkout() {

return "Run a hard 5k";

}

@Override

public String getDailyFortune() {

return "Just Do It: " + fortuneService.getFortune();

}

}

🔹 **Spring can create an instance using the no-arg constructor before injecting dependencies.**

**Class with Only Parameterized Constructor (Needs Explicit Configuration)**

java

CopyEdit

public class TrackCoach implements Coach {

private FortuneService fortuneService;

// No no-arg constructor provided

public TrackCoach(FortuneService fortuneService) {

this.fortuneService = fortuneService;

}

@Override

public String getDailyWorkout() {

return "Run a hard 5k";

}

@Override

public String getDailyFortune() {

return "Just Do It: " + fortuneService.getFortune();

}

}

🔹 **This class has no default constructor, so Spring requires explicit constructor injection configuration in XML or Java.**

**Solution: Explicit Constructor Injection in XML**

xml

CopyEdit

<bean id="myCoach" class="com.example.TrackCoach">

<constructor-arg ref="myFortuneService" />

</bean>

**Summary**

* **Spring prefers a no-arg constructor** to create beans easily.
* If only a **parameterized constructor exists**, **Spring needs explicit configuration**.
* No-arg constructors are required for **proxying, serialization, and frameworks like Hibernate**.

### ****FAQ: Why Use CricketCoach Instead of Coach Interface?****

**Q:** Why do we use the CricketCoach class instead of the Coach interface in setter injection?

**A:** The CricketCoach class has additional methods (getTeam() and setTeam()) that are not part of the Coach interface.

* If you retrieve the bean as Coach:

Coach theCricketCoach = context.getBean("myCricketCoach", Coach.class);

You can only access getDailyWorkout() and getDailyFortune().

* If you retrieve the bean as CricketCoach:

CricketCoach theCricketCoach = context.getBean("myCricketCoach", CricketCoach.class);

You can access **all methods**, including getTeam() and setTeam().

**Conclusion:** Your choice determines method visibility—use CricketCoach for full access or Coach for interface-level access.

### ****Important Notes on Init and Destroy Methods in XML Configuration****

When defining init-method and destroy-method in Spring XML, keep these points in mind:

1. **Access Modifier:** The method can be public, protected, or private.
2. **Return Type:** Any return type is allowed, but void is recommended since the return value is not used.
3. **Method Name:** You can choose any name for the method.
4. **Arguments:** The method **must not** accept any parameters (it should be a **no-arg** method).

### ****Special Note: Destroy Lifecycle and Prototype Scope****

In Spring, the **destroy-method** does **not** work for beans with **prototype scope**.

#### ****Why?****

* For **singleton beans**, Spring **manages the lifecycle**, including calling the destroy-method when the application shuts down.
* For **prototype beans**, Spring **only manages the creation** and does **not** track the bean after it is handed over.
* Since Spring does **not** manage the full lifecycle of prototype beans, it **does not call** the destroy-method automatically.

#### ****How to Handle Cleanup for Prototype Beans?****

* Manually call the cleanup method when you are done using the prototype bean.
* Use a **custom bean processor** or manage destruction using @PreDestroy and manually invoke it in your application.