✅ Spring Core Roadmap (Beginner to Intermediate)

🔹 1. Introduction to Spring Framework

- What is Spring?
- Why use Spring?
- Modules in Spring
- POJO, IoC, DI

🔹 2. Spring Bean Lifecycle

- What is a Bean?
- Bean Lifecycle Phases
- init-method & destroy-method
- BeanPostProcessor
- @PostConstruct and @PreDestroy

🔹 3. Dependency Injection (DI) in Spring

- Constructor Injection
- Setter Injection
- Field Injection
- @Autowired, @Qualifier
- XML vs Annotation-based DI

🔹 4. Configuration in Spring

- XML Configuration
- Annotation-based Configuration
- Java-based Configuration (@Configuration, @Bean)
- applicationContext.xml

🔹 **5. Bean Scopes in Spring**

- Singleton (default)
- Prototype
- Request, Session, GlobalSession (Web context)
- Scope using @Scope

🔹 **6. Autowiring in Spring**

- Types: no, byName, byType, constructor
- @Autowired, @Qualifier, @Primary
- Autowiring with interfaces

🔹 **7. Spring Expression Language (SpEL)**

- What is SpEL?
- Syntax and use
- Real use in annotation-based configuration

🔹 **8. Spring Profiles (Environment Specific Beans)**

- What are Profiles?
- Use of @Profile
- Activating profiles in XML and Java config
- Real project example (dev, prod)

🔹 **9. Spring AOP (Aspect Oriented Programming)**

- What is AOP?
- Joinpoint, Pointcut, Advice, Aspect
- @Aspect, @Before, @After, @Around

- Use cases: Logging, Security, Transactions

🔹 **10. Spring Annotations**

- @Component, @Service, @Repository, @Controller
- @Autowired, @Qualifier
- @Configuration, @Bean
- Lifecycle annotations: @PostConstruct, @PreDestroy

🔹 **11. ApplicationContext vs BeanFactory**

- Differences
- When to use what
- Examples

🔹 **12. Spring Event Handling**

- ApplicationEvent
- Custom Events
- @EventListener

🔹 **13. Internationalization (i18n) in Spring**

- MessageSource
- Loading messages for different locales

🔹 **14. Working with Properties in Spring**

- Using @Value
- Using PropertySourcesPlaceholderConfigurer
- External property files

🔹 15. Spring and Java 8+ Features

- Functional interfaces with Spring
- Optional and Lambda usage
- Stream API with Bean processing

🎯 Optional (For Spring Boot or Advanced Users)

Covered separately when we move to **Spring Boot**:

- Spring Boot AutoConfiguration
- Spring Boot Starter Projects
- Spring Boot Properties
- Spring Boot Profiles
- Spring Boot Testing

🧠 Final Interview Focus Areas from Roadmap:

Category	Must-Know Topics
Core Concepts	IoC, DI, Bean Lifecycle
Configuration	XML, Annotations, Java
Bean Management	Scopes, Autowiring
Advanced	AOP, Profiles, SpEL
Practical	ApplicationContext, Properties

Topic 1: Introduction to Spring Framework

🔷 What is the Spring Framework?

Spring is a powerful, lightweight, and open-source **Java framework** that simplifies **Java EE development** by providing **infrastructure support** for developing Java applications.

🔷 Why use Spring?

Feature	Benefit
✅ Loose Coupling	Through Dependency Injection (DI)
✅ Easy Testing	Due to POJO-based architecture
✅ Lightweight	No heavy application server required
✅ Modular	Pick what you need (Core, MVC, Boot, Security, etc.)
✅ Wide Adoption	Popular in enterprise applications
✅ Integration Support	Works well with Hibernate, JPA, MyBatis, etc.

🔷 Spring Modules

text

CopyEdit

Spring Framework

├— Core Container (Beans, Context, Core, Expression Language)

├— AOP (Aspect Oriented Programming)

├— JDBC and ORM

├— Web (Servlet, Web, Web MVC)

├— Testing (TestContext Framework)

You mainly use:

- **Core** → DI, Beans, Context

- **AOP** → Logging, Security
- **JDBC/ORM** → Data access
- **Web MVC** → Web apps

🔷 What are POJOs?

POJO = Plain Old Java Object

✅ Simple Java object with no special restrictions or dependencies.

In Spring, everything revolves around POJOs + configuration.

🔷 Inversion of Control (IoC)

🧠 **Definition**: Transferring the control of object creation and dependency binding **from your code to the Spring container**.

✅ Achieved using:

- **Dependency Injection (DI)**

🔷 Dependency Injection (DI)

Instead of a class creating its dependencies, Spring **injects them**.

✅ Example:

java

CopyEdit

```
class Student {
   private Address address;

   // Constructor or Setter
   public void setAddress(Address address) {
      this.address = address;
   }
}
```

In Spring, you configure Address in XML or annotations, and it auto-injects it into Student.

🔷 Ways to configure Spring

Method	Description
XML Configuration	Define beans in XML file (applicationContext.xml)
Annotation-based	Use @Component, @Autowired, etc.
Java-based	Use @Configuration, @Bean, etc.

🔷 What is a Spring Bean?

A **bean** is an object that is **managed by the Spring container**.

xml

CopyEdit

<bean id="student" class="com.example.Student"/>

🔷 Spring Container

The **Spring Container** is responsible for:

- Creating objects (beans)
- Injecting dependencies
- Managing lifecycle

Core Containers:

- BeanFactory (basic)
- ApplicationContext (advanced, preferred)

✅ Real-Life Example

💡 Think of Spring as a **restaurant**:

- You order food (you define what you need – configuration).

- Kitchen prepares it (Spring container creates beans).
- Waiter delivers it (DI injects into your class).
- You never enter the kitchen (you don't manage objects yourself).

🔷 Benefits of Spring Core

Benefit	Explanation
✅ Loose Coupling	Easy to manage and test
✅ Reusability	Beans can be reused
✅ Easy Unit Testing	Mock dependencies easily
✅ Modular	Only load what you need
✅ Easy Integration	With databases, APIs, etc.

🔷 Drawbacks

Drawback	Reason
❌ Steep learning curve	Especially for beginners
❌ Configuration overload	XML/Annotations can be complex
❌ Can over-engineer	Too many layers if not used wisely

🧠 Interview Questions

1. What is Spring Framework?
2. What is Inversion of Control (IoC)?
3. What is Dependency Injection?
4. What is a Spring Bean?
5. Difference between BeanFactory and ApplicationContext?
6. XML vs Annotation vs Java-based configuration?
7. Advantages of using Spring?

8. Can Spring work without a web server?

✅ Spring Core – Topic 2: Bean Lifecycle in Spring

🔷 What is a Bean in Spring?

A **bean** is simply a **Java object** that is managed by the **Spring container**. These objects are created, configured, and destroyed by Spring.

🔷 What is Bean Lifecycle?

The **Bean Lifecycle** is the **series of steps** that a bean goes through from its **creation to destruction**.
Spring provides **hooks** to perform actions at each step.

🌀 Complete Bean Lifecycle Flow

markdown

CopyEdit

1. Bean Instantiation (Object created)
2. Populate Properties (DI)
3. BeanNameAware
4. BeanFactoryAware / ApplicationContextAware
5. Pre-initialization (BeanPostProcessor#postProcessBeforeInitialization)
6. Custom init-method / @PostConstruct
7. Ready to use
8. Pre-destroy method / @PreDestroy
9. Destroy (custom destroy-method or DisposableBean)

🔹 Lifecycle Phases Explained

Phase	Description
Instantiation	Spring creates the object using constructor
Populate Properties	DI happens using setter/constructor
Awareness Interfaces	Optional - like BeanNameAware, ApplicationContextAware
Initialization	Custom logic like init-method, @PostConstruct
PostProcessors	Additional logic using BeanPostProcessor
Destruction	Cleanup using destroy-method, @PreDestroy

🔹 **Lifecycle Hooks (Ways to Hook Into Lifecycle)**

Type	Annotation / Interface
Custom Init	@PostConstruct, init-method, InitializingBean
Custom Destroy	@PreDestroy, destroy-method, DisposableBean
Lifecycle Handling	BeanPostProcessor, BeanFactoryAware, etc.

🔹 **Examples**

✅ **1. Using @PostConstruct and @PreDestroy**

java

CopyEdit

```
@Component
public class MyService {

  @PostConstruct
  public void init() {
    System.out.println("Bean initialized");
  }
```

```
    @PreDestroy
    public void destroy() {
        System.out.println("Bean will be destroyed");
    }
}
```

🔧 @PostConstruct runs **after the constructor and dependencies are injected**.
🧹 @PreDestroy runs **just before Spring destroys the bean**.

✅ 2. Using XML (init-method and destroy-method)

xml

CopyEdit

```
<bean id="myBean" class="com.example.MyBean"
    init-method="customInit" destroy-method="customDestroy"/>
```

🟢 Real-Life Analogy

💡 Think of a bean as a **robot** built in a factory:

- The robot is built (instantiated)
- It is programmed (property injection)
- It goes through a final check (init)
- It performs tasks
- Then it shuts down properly (destroy)

✅ Benefits

Benefit	Why it matters
Custom init/destroy	Control over bean setup/cleanup
Test lifecycle steps	Know exactly when what runs
Plug in logic	Auto logging, validation during bean creation

❌ Drawbacks

Drawback	Solution
Complex lifecycle	Use annotations like @PostConstruct to simplify
Overkill for simple beans	Avoid using all hooks unless needed

---

🧠 Interview Questions

1. What is a Spring Bean?
2. Explain the Bean Lifecycle in Spring.
3. What are @PostConstruct and @PreDestroy?
4. What is BeanPostProcessor?
5. Difference between init-method and @PostConstruct?
6. When does destroy method run?

✅ Spring Core – Topic 3: Dependency Injection (DI) in Spring

🔷 What is Dependency Injection?

Dependency Injection (DI) is a **design pattern** used to reduce **tight coupling** between classes. Instead of a class creating its dependencies, those dependencies are provided (injected) by a **container** (in Spring's case, the ApplicationContext).

🔷 Why use Dependency Injection?

Problem Without DI	Solution With DI
Hard-coded dependencies	Inject dependencies externally
Difficult to test	Easily mock dependencies
Tight coupling	Loose coupling

🔷 Types of Dependency Injection in Spring

Spring supports 3 types of DI:

Type	Description
1. **Constructor Injection**	Dependencies are injected via class constructor
2. **Setter Injection**	Dependencies are injected via public setter methods
3. **Field Injection** *(Not recommended for testing)*	Directly injects fields using @Autowired

---

✅ 1. Constructor Injection (Recommended)

java

CopyEdit

@Component

public class StudentService {

```
    private final StudentRepository repository;

    @Autowired
    public StudentService(StudentRepository repository) {
        this.repository = repository;
    }
}
```

✅ Best for **immutability**, **testing**, and ensuring **required dependencies**.

✅ **2. Setter Injection**

```
java
CopyEdit
@Component
public class StudentService {

    private StudentRepository repository;

    @Autowired
    public void setRepository(StudentRepository repository) {
        this.repository = repository;
    }
}
```

✅ Use when **optional dependency** or **circular dependency** exists.

✅ **3. Field Injection (Not Ideal for Unit Testing)**

```
java
CopyEdit
```

```
@Component
public class StudentService {

  @Autowired
  private StudentRepository repository;
}
```

⚠️ Avoid if possible. It tightly couples Spring with your code and makes testing harder.

🔷 @Autowired

- Used for automatic injection.
- By default, it injects by **type**.
- Can be used on **constructor, setter, or field**.

🔷 @Qualifier

Used when **multiple beans of the same type** are present.

```
java
CopyEdit
@Autowired
@Qualifier("scienceTeacher")
private Teacher teacher;
```

🔷 @Primary

Used to mark a **default bean** if multiple types exist.

```
java
CopyEdit
@Primary
@Bean
public Teacher defaultTeacher() {
```

```
    return new MathTeacher();
}
```

🔷 XML Configuration Example

xml

CopyEdit

```
<bean id="studentService" class="com.app.StudentService">
    <property name="repository" ref="studentRepository"/>
</bean>

<bean id="studentRepository" class="com.app.StudentRepository"/>
```

🔷 Real-Life Analogy

💡 **DI is like getting groceries delivered**:

- Without DI: You go to the store, pick everything.
- With DI: Groceries are delivered to your door (you just use them).

✅ Benefits of DI

Benefit	Explanation
✅ Loose Coupling	Easy to swap implementations
✅ Reusability	Same bean in multiple places
✅ Easy Testing	Inject mocks/stubs
✅ Easy Maintenance	Focus on logic, not plumbing

❌ Drawbacks

Drawback	Solution
Too many beans	Organize packages and use @ComponentScan
Hidden wiring	Use Java-based configuration for clarity
Constructor injection becomes large	Refactor into smaller components

---

🧠 Interview Questions

1. What is Dependency Injection?
2. What are the types of DI in Spring?
3. Which DI is preferred and why?
4. What is the difference between @Autowired, @Qualifier, and @Primary?
5. Constructor vs Setter injection?
6. Can we inject one bean into another? How?
7. How does DI help in writing testable code?

✅ Spring Core – Topic 4: Spring Configuration (XML, Annotations, Java-based)

🔷 What is Spring Configuration?

Spring configuration tells the **Spring container how to create, manage, and wire** your beans. It can be done using:

1. **XML Configuration** (Traditional)
2. **Annotation-based Configuration** (Modern)
3. **Java-based Configuration** (Highly preferred)

🔷 Why Do We Need Spring Configuration?

Without configuration, Spring doesn't know:

- What beans exist
- How to inject dependencies
- When to initialize or destroy beans

Spring configuration allows:

- Clean separation of concerns
- Centralized bean management
- Flexible dependency injection

✅ 1. XML-Based Configuration (Older Style)

🔹 XML Example:

xml

CopyEdit

```
<!-- applicationContext.xml -->
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
      http://www.springframework.org/schema/beans
```

```
        http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="studentService" class="com.app.StudentService">
        <property name="repository" ref="studentRepository"/>
    </bean>

    <bean id="studentRepository" class="com.app.StudentRepository"/>
</beans>
```

📌 You load it using:

```
java
CopyEdit
ApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext.xml");
```

✅ **2. Annotation-Based Configuration (Modern Style)**

🔹 **Java Class with Annotations**

```
java
CopyEdit
@Component
public class StudentRepository {
    // Logic
}

@Service
public class StudentService {
    @Autowired
    private StudentRepository repository;
}
```

📌 Annotations used:

- @Component, @Service, @Repository, @Controller
- @Autowired for injection

📦 Auto-scan the components:

java

CopyEdit

```
AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext("com.app");
```

✅ 3. Java-Based Configuration (Best Practice)

Instead of XML, we use **Java classes** with @Configuration and @Bean.

🔹 **Example:**

java

CopyEdit

```
@Configuration
public class AppConfig {

  @Bean
  public StudentRepository studentRepository() {
    return new StudentRepository();
  }

  @Bean
  public StudentService studentService() {
    return new StudentService(studentRepository());
  }
}
```

📌 Load it like:

java

CopyEdit

```
AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext(AppConfig.class);
```

🔍 Comparison Table

Feature	XML-Based	Annotation-Based	Java-Based
Verbose	Yes	No	No
Readable	Moderate	High	High
Type-safe	❌	✅	✅
Refactor-friendly	❌	✅	✅
Popular Today	Rare	Very Common	Very Common

✅ Benefits

Benefit	Description
Centralized Control	Easy to manage app settings
Flexible	Can mix XML + Java + Annotations
Scalable	Easy to maintain large apps
Type-safe	Java config gives compile-time safety

❌ Drawbacks

Drawback	Solution
XML is verbose	Use annotations or Java-based config
Hidden wiring in annotations	Use explicit Java-based configuration

---

🧠 Interview Questions

1. How many types of configuration are there in Spring?
2. Which configuration is best and why?
3. What is the difference between @Bean and @Component?
4. Can you mix XML and annotation config?
5. What is the purpose of @Configuration and @ComponentScan?
6. How does Java-based config improve maintainability?

✅ Spring Core – Topic 5: Bean Scopes in Spring

🔷 What is Bean Scope?

Bean scope in Spring defines **how many instances** of a bean are created and **how** they are shared or isolated across requests.

In simple words:
Bean Scope = Bean Life Span + Visibility + Instance Count

🔷 Types of Bean Scopes in Spring

Scope Name	Description	Scope Type
singleton	Single instance per Spring container	Default (Global)
prototype	New instance every time it's requested	Local
request	One instance per HTTP request (web apps only)	Web
session	One instance per HTTP session (web only)	Web
application	One instance per ServletContext	Web
websocket	One per WebSocket session	Web

✅ 1. Singleton (Default)

- Spring creates **only one instance** per container.
- Used for **stateless services**.

java

CopyEdit

```
@Component
@Scope("singleton") // Optional, because it's default
public class MyBean { }
```

✅ 2. Prototype

- Spring creates a **new instance every time** the bean is requested.
- Used when each user or thread needs a new object.

java

CopyEdit

@Component

@Scope("prototype")

public class MyPrototypeBean { }

✅ **3. Request**

- One bean per **HTTP request**.
- Used in web applications for request-specific data.

java

CopyEdit

@Component

@Scope("request")

public class RequestScopedBean { }

✅ **4. Session**

- One bean per **HTTP session**.
- Used when data must persist across multiple requests from the same user.

java

CopyEdit

@Component

@Scope("session")

public class SessionScopedBean { }

✅ **5. Application**

- One bean per **ServletContext**.

- Shared across the entire application, like a singleton but at servlet level.

java

CopyEdit

@Component

@Scope("application")

public class AppScopeBean { }

✅ **6. WebSocket (Less Common)**

- One bean per WebSocket connection/session.

java

CopyEdit

@Component

@Scope("websocket")

public class WebSocketScopedBean { }

🧠 **Real-Life Analogy**

Scope	Real World Example
Singleton	CEO – only one in the company
Prototype	Employee – new ID each time you hire
Request	Customer visit – valid only during visit
Session	Logged-in user – valid during login session
Application	App-wide setting – same for all users

✅ **Benefits of Scopes**

Scope	When to Use
Singleton	Stateless services like logging, DAO, services

Scope	When to Use
Prototype	Stateful beans like form data, worker tasks
Request	Form binding, request-specific beans in MVC
Session	User-specific data in web apps
Application	Shared configuration or static resources

❌ Drawbacks and Cautions

Scope	Issue	Caution
Prototype	Not managed after creation	You must manage its lifecycle manually
Singleton	Shared across threads	Avoid mutable shared state
Session/Request	Only in web apps	Will throw error in standalone

---

🧪 Example Using Java Config

java

CopyEdit

```
@Bean
@Scope("prototype")
public MyBean prototypeBean() {
  return new MyBean();
}
```

🧠 Interview Questions

1. What is the default scope of a Spring bean?
2. What is the difference between Singleton and Prototype?
3. When would you use a session-scoped bean?
4. Are request/session-scoped beans available in a console app?

5. How are prototype beans destroyed?

✅ Spring Core – Topic 6: Autowiring in Spring

🔷 What is Autowiring?

Autowiring is a feature in Spring that allows the **Spring container to automatically inject dependencies** into a bean without using explicit @Bean or XML configuration for wiring.

👉 It simplifies configuration and makes your code cleaner and less boilerplate.

🔷 Why Use Autowiring?

Without Autowiring	With Autowiring
You write manual wiring code	Spring auto-detects and injects
More configuration required	Less configuration
Tight coupling possible	Promotes loose coupling

🔷 How Does Autowiring Work?

Spring tries to **match the type** (and sometimes the name) of a dependency and **injects it automatically**.

🔷 Autowiring Modes

Mode	Description
byType	Matches by data type
byName	Matches by property name
constructor	Injects using constructor
autodetect *(deprecated)*	Tries constructor, then byType

🔸 These are mostly used in **XML config**.

✅ **Autowiring Using Annotations**

🔹 **1. @Autowired (Most Common)**

Spring's most common autowiring annotation.

java

CopyEdit

```
@Component
public class StudentService {

  @Autowired
  private StudentRepository repository;
}
```

Spring will:

- Look for a bean of type StudentRepository
- Inject it automatically

🔹 **2. @Autowired with Constructor (Recommended)**

java

CopyEdit

```
@Component
public class StudentService {

  private final StudentRepository repository;

  @Autowired
  public StudentService(StudentRepository repository) {
    this.repository = repository;
```

```
    }
}
```

✅ Most testable
✅ Ensures required dependencies
✅ Good for immutability

🔹 3. @Autowired with Setter

java

CopyEdit

```
@Component
public class StudentService {

    private StudentRepository repository;

    @Autowired
    public void setRepository(StudentRepository repository) {
        this.repository = repository;
    }
}
```

✅ Useful if the dependency is optional
✅ Supports circular dependencies

🔹 4. @Autowired(required = false)

Used when the dependency may or may not be present.

java

CopyEdit

```
@Autowired(required = false)
private OptionalService service;
```

🔹 5. @Qualifier

Used when there are **multiple beans of the same type**.

java

CopyEdit

@Autowired

@Qualifier("mathTeacher")

private Teacher teacher;

🔹 6. @Primary

Specifies the **default bean** when multiple beans of the same type exist.

java

CopyEdit

@Primary

@Bean

public Teacher defaultTeacher() {

  return new MathTeacher();

}

✅ Real-Life Analogy

📦 Think of autowiring like ordering a food combo:

- You ask for a “Veg Combo”
- The system automatically adds rice + dal + roti based on the type

You didn’t choose each item — the system knew what to put.

✅ Benefits

Benefit	Description
✅ Less Boilerplate	No need to write getter/setter wiring
✅ Easier to Maintain	Just declare and use
✅ Promotes Loose Coupling	Interface-based injection
✅ Testable	Constructor injection supports unit testing

❌ Drawbacks

Drawback	Solution
Multiple Beans Cause Confusion	Use @Qualifier
Autowiring Fails Silently	Use @Autowired(required = true)
Hard to Read	Prefer constructor-based wiring for clarity

🧠 Interview Questions

1. What is Autowiring in Spring?
2. What are the types of autowiring?
3. What is the difference between constructor and field injection?
4. How to handle multiple beans of the same type?
5. What happens if no bean is found while autowiring?
6. Is autowiring mandatory in Spring?

✅ Spring Core – Topic 7: Spring Bean Lifecycle

🔷 What is the Spring Bean Lifecycle?

In Spring, every bean has a **lifecycle** — from creation to destruction. Spring manages the lifecycle using **callback methods**, **interfaces**, and **annotations**.

This lifecycle is controlled by the **Spring container** and includes:

1. Instantiation
2. Property injection
3. Custom initialization
4. Usage
5. Custom destruction

🔷 Why Understand Bean Lifecycle?

- Helps you execute custom logic **before or after** the bean is fully initialized.
- Useful in **resource management** (e.g., DB connection, file open/close).
- Helps in debugging and optimization.

✅ Stages in Spring Bean Lifecycle

Stage	Description
1. Instantiation	Spring creates the bean using the constructor
2. Populate Properties	Dependencies are injected (@Autowired, etc.)
3. Set Bean Name	Spring sets the bean name (BeanNameAware)
4. Set Bean Factory	Set by Spring (BeanFactoryAware)
5. Pre-Initialization	BeanPostProcessor.postProcessBeforeInitialization()
6. Custom Init	User-defined init method or @PostConstruct
7. Ready to Use	Bean is in use

Stage	Description
8. Pre-Destruction	@PreDestroy, or destroy-method
9. Destroy	Bean is destroyed by container

✅ **Ways to Interact with the Lifecycle**

🔹 **1. Using @PostConstruct and @PreDestroy**

java

CopyEdit

```
@Component
public class MyBean {

  @PostConstruct
  public void init() {
    System.out.println("Bean is initialized");
  }

  @PreDestroy
  public void cleanup() {
    System.out.println("Bean is about to be destroyed");
  }
}
```

🔹 **2. Using InitializingBean and DisposableBean interfaces**

java

CopyEdit

```
@Component
```

```
public class MyBean implements InitializingBean, DisposableBean {

    @Override
    public void afterPropertiesSet() {
        System.out.println("Init via InitializingBean");
    }

    @Override
    public void destroy() {
        System.out.println("Destroy via DisposableBean");
    }
}
```

🔹 3. Custom Init and Destroy Methods (Java Config or XML)

java

CopyEdit

```
@Bean(initMethod = "init", destroyMethod = "cleanup")
public MyBean myBean() {
    return new MyBean();
}
```

Or in XML:

xml

CopyEdit

```
<bean id="myBean" class="com.MyBean" init-method="init" destroy-method="cleanup"/>
```

🔹 4. Using BeanPostProcessor for Global Hooks

java

CopyEdit

```
@Component
public class MyPostProcessor implements BeanPostProcessor {

  public Object postProcessBeforeInitialization(Object bean, String beanName) {
    System.out.println("Before init: " + beanName);
    return bean;
  }

  public Object postProcessAfterInitialization(Object bean, String beanName) {
    System.out.println("After init: " + beanName);
    return bean;
  }
}
```

✅ **Real-Life Analogy**

Imagine a new employee in a company:

- Constructor → Hiring
- @Autowired → Providing resources
- @PostConstruct → Orientation
- Working → Regular work
- @PreDestroy → Exit interview
- Destroy → Employee leaves company

✅ **Benefits**

Benefit	Description
Initialization control	You can run custom setup logic

Benefit	Description
Resource handling	Open/close DB or files properly
Better lifecycle management	Hook into lifecycle for logging, security etc.

❌ Drawbacks / Things to Watch

Concern	Detail
Overusing lifecycle hooks	Can make code hard to test
Mixing too many approaches	Pick one (annotations or interface) for consistency
Not closing resources	Always use destroy methods for cleanup

🧠 Interview Questions

1. What is the bean lifecycle in Spring?
2. How can you hook into initialization or destruction?
3. What is the difference between @PostConstruct and afterPropertiesSet()?
4. When is BeanPostProcessor used?
5. What are alternatives to @PreDestroy?

✅ Spring Core – Topic 8: Dependency Injection (DI) in Spring

🔷 What is Dependency Injection?

Dependency Injection (DI) is a design pattern where **an object's dependencies are provided from the outside**, instead of the object creating them itself.

🔑 In simple terms:

Rather than a class creating the object it needs, the Spring container **injects it** for you.

🔷 Why is Dependency Injection Important?

Without DI	With DI (Spring)
Class creates its own dependencies	Dependencies are injected externally
Tight coupling	Loose coupling
Hard to test or reuse	Easy to test and swap components

✅ Types of Dependency Injection in Spring

Type	Description
Constructor DI	Dependencies passed via constructor
Setter DI	Dependencies injected via setters
Field DI	Direct injection into class fields

🔹 1. Constructor Injection (Recommended ✅)

java

CopyEdit

```
@Component
public class StudentService {
   private final StudentRepository repo;
```

```
    @Autowired
    public StudentService(StudentRepository repo) {
        this.repo = repo;
    }
}
```

✅ Best for immutability
✅ Clear dependencies
✅ Easy to test

🔹 2. Setter Injection

```
java
CopyEdit
@Component
public class StudentService {
    private StudentRepository repo;

    @Autowired
    public void setRepo(StudentRepository repo) {
        this.repo = repo;
    }
}
```

✅ Good for optional dependencies
✅ Allows changing dependency at runtime

🔹 3. Field Injection (Not Recommended ❌)

```
java
CopyEdit
@Component
```

```
public class StudentService {

  @Autowired
  private StudentRepository repo;
}
```

❌ Harder to test
❌ Violates encapsulation

✅ **Real-Life Analogy**

Let's say you run a coffee shop:

- Without DI: The barista goes out, buys beans, machines, cups, etc.
- With DI: You provide everything (beans, cups, tools) to the barista.

Result: Barista focuses on **making coffee**, not managing resources.

✅ **Benefits of Dependency Injection**

Benefit	Description
✅ Loose Coupling	Classes don't depend on implementations directly
✅ Easy Testing	Use mocks/fakes easily in unit tests
✅ Reusable Code	Swap components without changes
✅ Cleaner Code	Delegates object creation to Spring
✅ Better Maintenance	Less code repetition

❌ **Drawbacks / Considerations**

Drawback	How to Handle
Too many dependencies	Break the class into smaller ones
Circular dependencies	Avoid or refactor

Drawback	How to Handle
Hidden field injection	Prefer constructor injection for clarity

---

✅ DI with Java Configuration

java

CopyEdit

```
@Configuration
public class AppConfig {

  @Bean
  public StudentService studentService() {
    return new StudentService(studentRepository());
  }

  @Bean
  public StudentRepository studentRepository() {
    return new StudentRepository();
  }
}
```

🧠 Interview Questions

1. What is Dependency Injection?
2. What are the different types of DI in Spring?
3. Which is better: constructor or field injection? Why?
4. What happens if multiple beans of the same type are available?
5. How does DI improve testability?

✅ Spring Core – Topic 9: Common Spring Annotations (Overview with Uses)

🔷 What are Spring Annotations?

Spring uses **annotations** to reduce XML configuration and make the application **more readable and manageable**. Annotations tell the Spring container **what to do**, such as:

- Which class is a bean?
- Where to inject dependencies?
- How to map web requests?

✅ Core Spring Annotations (Used in Spring Core)

🔹 1. @Component

Marks a Java class as a **Spring-managed bean**.

java

CopyEdit

@Component

public class StudentService { }

- Automatically detected during component scanning
- Used for **generic beans**

🔹 2. @Service

Specialized form of @Component used in the **service layer**.

java

CopyEdit

@Service

public class StudentService { }

- Indicates business logic component
- Helps with **better readability**

🔹 3. @Repository

Used for **DAO classes** that interact with the database.

java

CopyEdit

@Repository

public class StudentRepository { }

- Catches persistence exceptions and rethrows them as Spring exceptions

---

🔹 4. @Controller

Used in **Spring MVC** to mark a class as a **web controller**.

java

CopyEdit

@Controller

public class StudentController { }

- Handles HTTP requests
- Works with JSP or Thymeleaf views

🔹 5. @RestController

Combination of @Controller and @ResponseBody.

java

CopyEdit

@RestController

public class StudentAPI { }

- Used for RESTful APIs
- Automatically returns JSON or XML

🔹 6. @Autowired

Automatically injects a dependency.

java

CopyEdit

@Autowired

private StudentRepository repo;

- Can be used with constructor, setter, or field

🔹 7. @Qualifier

Used to **resolve conflicts** when multiple beans of the same type exist.

java

CopyEdit

@Autowired

@Qualifier("mathTeacher")

private Teacher teacher;

🔹 8. @Primary

Marks one bean as **default** when multiple beans of the same type exist.

java

CopyEdit

@Primary

@Bean

public Teacher defaultTeacher() {

  return new MathTeacher();

}

🔹 9. @Value

Injects values from application.properties.

java

CopyEdit

@Value("${app.name}")

private String appName;

🔹 10. @Scope

Specifies bean scope: singleton, prototype, etc.

java

CopyEdit

@Scope("prototype")

@Component

public class TempBean { }

🔹 11. @PostConstruct and @PreDestroy

Used to run **initialization** and **cleanup** code.

java

CopyEdit

@PostConstruct

public void init() { }

@PreDestroy

public void destroy() { }

✅ Real-Life Analogy

Think of annotations like **labels** on packages:

- @Service: This package contains tools
- @Repository: This package contains spare parts
- @Autowired: Attach this spare part to the tool
- @RestController: This is a delivery section for REST orders

---

✅ Benefits of Annotations

Benefit	Description
✅ Less XML	Replaces complex XML configuration
✅ Readable	Easy to understand what a class does
✅ Flexible	Fine-grained control
✅ Cleaner Code	Clear structure & behavior

❌ Drawbacks / Things to Consider

Concern	Solution
Overuse can hide behavior	Use clearly and document properly
Too many annotations = confusion	Use separation of layers (Controller, Service, Repository)
Implicit wiring	Prefer constructor injection for clarity

🧠 Interview Questions

1. What is the use of @Component, @Service, @Repository?
2. Difference between @Controller and @RestController?
3. How does @Autowired work?
4. When to use @Qualifier and @Primary?
5. Can we create custom annotations in Spring?

✅ Spring Core – Topic 10: Spring Configuration (XML vs Java vs Annotations)

🔷 What is Spring Configuration?

Spring Configuration refers to **how you define and register beans** in the Spring Container so that they can be managed and injected as needed.

✅ There are 3 ways to configure Spring Beans:

Method	Description
1. XML-based	Traditional way using XML files
2. Annotation-based	Uses annotations like @Component
3. Java-based	Uses Java classes with @Configuration

🔹 1. XML-Based Configuration

Beans are defined inside an XML file, usually applicationContext.xml.

xml

CopyEdit

<bean id="studentService" class="com.example.StudentService"/>

<bean id="studentRepository" class="com.example.StudentRepository"/>

✅ Benefits:

- Clear separation of config and code
- Good for legacy projects

❌ Drawbacks:

- Verbose and hard to maintain
- Not type-safe
- No IDE support for refactoring

🔹 2. Annotation-Based Configuration (Modern ✅)

Uses annotations to mark beans and inject dependencies.

java

CopyEdit

@Component

public class StudentService { }

@Service

public class StudentRepository { }

@Autowired

private StudentRepository repository;

You must enable component scanning:

xml

CopyEdit

<context:component-scan base-package="com.example"/>

Or in Java config:

java

CopyEdit

@ComponentScan("com.example")

✅ **Benefits:**

- Concise and readable
- Less configuration
- Encourages layered architecture

❌ **Drawbacks:**

- Hidden wiring (if overused)
- Needs careful design

🔹 **3. Java-Based Configuration (Recommended ✅)**

Use @Configuration classes to define beans:

java

CopyEdit

```
@Configuration
public class AppConfig {

  @Bean
  public StudentService studentService() {
    return new StudentService(studentRepository());
  }

  @Bean
  public StudentRepository studentRepository() {
    return new StudentRepository();
  }
}
```

✅ **Benefits:**

- Type-safe, refactor-friendly
- Full power of Java (conditions, logic)
- Clean and structured

❌ **Drawbacks:**

- Slightly more verbose than annotations
- Still requires understanding of annotations

✅ **Which One to Use?**

Use Case	Best Choice
New project	Java + Annotations ✅

Use Case	Best Choice
Large enterprise w/ legacy XML	XML or mixed
Test environments	Java-based config
Need logic in bean creation	Java-based config ✅

✅ Real-Life Analogy

Think of this like setting up a team in your company:

- **XML config**: A detailed manual document that lists everyone's roles and relationships.
- **Annotations**: Sticky notes placed on people's desks indicating what they do.
- **Java Config**: A project manager writes Java code that sets up roles based on logic and rules.

🧠 Interview Questions

1. What are the ways to configure beans in Spring?
2. What is the difference between XML and annotation-based configuration?
3. What are the benefits of Java-based configuration?
4. Can you use multiple configuration types together?
5. Which configuration style is preferred in modern applications?

✅ Spring Core – Topic 11: Bean Scopes in Spring Framework

🔷 What is a Bean Scope?

A **Bean Scope** defines **how many instances** of a bean are created and **how long they live** in the Spring container.

In simple terms:

Bean scope = lifecycle + visibility of the bean.

✅ Supported Bean Scopes in Spring (Core)

Scope	Description
singleton	One shared instance per Spring container (default)
prototype	New instance every time bean is requested
request	One instance per HTTP request (Spring MVC/Web only)
session	One instance per HTTP session (Web only)
application	One instance per ServletContext (Web only)
websocket	One instance per WebSocket session

🔹 1. singleton (Default)

java

CopyEdit

@Scope("singleton")

@Component

public class StudentService { }

- One instance for the entire Spring container
- Best for stateless services

✅ **Benefit**: memory efficient
❌ **Drawback**: not suitable for stateful operations

🔹 **2. prototype**

java

CopyEdit

@Scope("prototype")

@Component

public class TempService { }

- A new object is created each time you request the bean
- Good for stateful, temporary use cases

✅ Used when each object has different data
❌ Spring doesn't manage lifecycle after creation

🔹 **3. request (Web Only)**

java

CopyEdit

@Scope("request")

@Component

public class RequestBean { }

- One bean instance per **HTTP request**
- Useful for request-specific data (e.g., tracking user action)

🔹 **4. session (Web Only)**

java

CopyEdit

@Scope("session")

@Component

public class SessionBean { }

- One bean per **user session**

- Good for login sessions or user preferences

🔹 **5. application (Web Only)**

java

CopyEdit

@Scope("application")

@Component

public class AppScopeBean { }

- One bean instance for the **whole application (ServletContext)**

🔹 **6. websocket (Web Only)**

java

CopyEdit

@Scope("websocket")

@Component

public class WebSocketBean { }

- One bean instance per **WebSocket connection**

✅ **How to Declare Bean Scope**

➤ **Using Annotations:**

java

CopyEdit

@Component

@Scope("prototype")

public class MyBean { }

➤ **Using Java Configuration:**

java

CopyEdit

```
@Bean
@Scope("prototype")
public MyBean myBean() {
   return new MyBean();
}
```

✅ **Real-Life Analogy**

Imagine you're running a **bike rental service**:

- **singleton** → 1 shared manager (one instance for everyone)
- **prototype** → Each customer gets a new bike (new instance each time)
- **request** → New map for each customer's ride (one per request)
- **session** → Personal locker for each customer session
- **application** → Common announcement board (shared across app)

✅ **Interview Questions**

1. What is the default scope in Spring?
2. What is the difference between singleton and prototype scope?
3. When should you use request/session scope?
4. Is Spring responsible for destroying prototype beans?
5. Can you define custom scopes?

✅ Spring Core – Topic 12: Spring Bean Lifecycle (Initialization and Destruction)

🔷 What is Bean Lifecycle?

The **bean lifecycle** in Spring defines how a bean is:

1. **Created**
2. **Initialized**
3. **Used**
4. **Destroyed**

Spring gives you **hooks (methods)** to insert custom logic at different lifecycle stages.

✅ Lifecycle Phases

1. **Bean Instantiation** – Object is created using new
2. **Populate Properties** – Dependencies are injected
3. **Bean Name Set** – Name is assigned by Spring container
4. **Aware Interfaces Called** – BeanNameAware, ApplicationContextAware etc.
5. **Pre-initialization** – BeanPostProcessor's postProcessBeforeInitialization()
6. **Custom Init Method** – Your init-method or @PostConstruct
7. **Ready to Use**
8. **Pre-destroy** – @PreDestroy or destroy-method runs
9. **Bean Destroyed** – Bean is removed from the container

✅ Ways to Hook into Lifecycle

🔹 1. Using @PostConstruct and @PreDestroy

java

CopyEdit

@Component

```
public class MyBean {

  @PostConstruct
  public void init() {
    System.out.println("Bean initialized");
  }

  @PreDestroy
  public void cleanup() {
    System.out.println("Bean destroyed");
  }
}
```

✅ Recommended
❌ Only works for **singleton** scoped beans

🔹 **2. Using InitializingBean and DisposableBean interfaces**

```
java
CopyEdit
@Component
public class MyBean implements InitializingBean, DisposableBean {

  @Override
  public void afterPropertiesSet() throws Exception {
    System.out.println("Bean initialized");
  }

  @Override
  public void destroy() throws Exception {
```

```
    System.out.println("Bean destroyed");
  }
}
```

✅ Good for core Spring control
❌ Tightly couples your class with Spring

🔹 3. Using XML config (init-method and destroy-method)

xml

CopyEdit

```
<bean id="myBean" class="com.example.MyBean"
    init-method="init" destroy-method="cleanup"/>
```

In Java config:

java

CopyEdit

```
@Bean(initMethod = "init", destroyMethod = "cleanup")
public MyBean myBean() {
  return new MyBean();
}
```

✅ Flexible
❌ Old approach (not preferred in modern apps)

🔁 BeanPostProcessor (Advanced)

Intercepts bean lifecycle for additional logic.

java

CopyEdit

```
@Component
public class CustomBeanPostProcessor implements BeanPostProcessor {
```

```
    public Object postProcessBeforeInitialization(Object bean, String name) {
        // logic before init
        return bean;
    }

    public Object postProcessAfterInitialization(Object bean, String name) {
        // logic after init
        return bean;
    }
}
```

✅ Useful for cross-cutting concerns like logging or security

✅ Real-Life Analogy

Think of creating a **startup office**:

- @PostConstruct → You turn on the lights and welcome employees
- @PreDestroy → You shut down the office at the end of the day
- InitializingBean → Manual steps done by office manager
- BeanPostProcessor → Security checks before and after staff enter

---

✅ Interview Questions

1. What are different ways to define init and destroy methods in Spring?
2. What is the use of @PostConstruct and @PreDestroy?
3. Difference between InitializingBean and @PostConstruct?
4. When is BeanPostProcessor used?
5. Can @PreDestroy be used with prototype scope?

✅ Spring Core – Topic 13: Dependency Injection (DI) – Constructor vs Setter

---

🔷 What is Dependency Injection?

Dependency Injection (DI) is a design pattern used in Spring to **automatically provide (inject) required objects (dependencies)** to a class.

✅ Why We Use DI?

- **Loose coupling**
- **Easy testing**
- **Better code maintainability**
- **Flexibility in configuration**

✅ Types of DI in Spring

Type	Description
Constructor DI	Dependencies are passed via constructor
Setter DI	Dependencies are set via setter methods

🔹 Constructor Injection (Recommended ✅)

java

CopyEdit

```
@Component
public class StudentService {
  private final StudentRepository repo;

  @Autowired
  public StudentService(StudentRepository repo) {
    this.repo = repo;
```

```
    }
}
```

✅ **Benefits**:

- Mandatory dependencies are clearly defined
- Good for **immutability**
- Makes testing easy

🔹 **Setter Injection**

```
java
CopyEdit
@Component
public class StudentService {

    private StudentRepository repo;

    @Autowired
    public void setStudentRepository(StudentRepository repo) {
        this.repo = repo;
    }
}
```

✅ Useful when:

- Dependency is **optional**
- You need to reassign values later

✅ **Which One Should You Use?**

Criteria	Best Choice
Mandatory dependency	Constructor ✅

Criteria	Best Choice
Optional dependency	Setter
Immutability needed	Constructor ✅
Legacy / flexible config	Setter

✅ Real-Life Analogy

- **Constructor DI**: Hiring someone with all skills upfront — ready to work.
- **Setter DI**: Hiring and training them later as needed.

✅ Interview Questions

1. What is dependency injection?
2. What is the difference between constructor and setter injection?
3. Which injection is preferred and why?
4. Can you mix both types in the same class?

✅ Spring Core – Topic 14: Spring Stereotype Annotations

🔷 What Are Stereotype Annotations?

In Spring, **stereotype annotations** are special annotations used to **declare and register beans automatically** during component scanning.

✅ Common Annotations

Annotation	Purpose
@Component	Generic Spring-managed bean
@Service	Marks service layer class
@Repository	Marks DAO layer class (data access logic)
@Controller	Marks Spring MVC controller (used in web apps)

🔹 1. @Component

- Generic bean
- Spring detects and registers it during scanning

java

CopyEdit

@Component

public class MyUtilityClass { }

✅ Used when no specific layer fits

🔹 2. @Service

- Indicates **service logic/business logic** class

java

CopyEdit

@Service

```
public class StudentService { }
```

✅ Helps organize layers
✅ Better readability in large apps

🔹 **3. @Repository**

- Indicates **DAO (Data Access Object)** class

java

CopyEdit

```
@Repository
public class StudentRepository {
  // DB access code
}
```

✅ Also enables **automatic exception translation** (e.g., converts DB errors to Spring DataAccessException)

🔹 **4. @Controller (Spring MVC only)**

- Indicates a **web controller** (handles HTTP requests)

java

CopyEdit

```
@Controller
public class StudentController {

  @GetMapping("/home")
  public String homePage() {
    return "home";
  }
}
```

✅ Used in **Spring MVC web apps**
✅ Automatically maps HTTP requests

🔹 **BONUS: @RestController (Spring Web Only)**

java

CopyEdit

```
@RestController
public class ApiController {

  @GetMapping("/api/data")
  public String getData() {
    return "Hello from API!";
  }
}
```

✅ Combines @Controller + @ResponseBody
✅ Used in REST APIs to return JSON/XML

✅ **Component Scanning Setup**

If you're using these annotations, Spring needs to scan the package:

java

CopyEdit

```
@Configuration
@ComponentScan("com.example")
public class AppConfig { }
```

✅ **Real-Life Analogy**

Think of your app like a **company**:

- @Component → General staff

- @Service → Operations department
- @Repository → Accounts/Records department
- @Controller → Front desk (handles user requests)

✅ **Interview Questions**

1. What is the difference between @Component, @Service, and @Repository?
2. Why use @Repository instead of @Component?
3. What is @RestController and how is it different from @Controller?
4. How does component scanning work in Spring?

✅ Spring Core – Topic 15: Autowiring in Spring (@Autowired, @Qualifier)

🔷 What is Autowiring?

Autowiring is Spring's way of **automatically injecting dependencies** into a class without needing to manually set them.

It reduces boilerplate and makes your code cleaner.

✅ Why Use Autowiring?

- Removes manual new object creation
- Enables **loose coupling**
- Makes code **easier to test and maintain**

🔹 Example without Autowiring:

java

CopyEdit

```
public class StudentService {
   private StudentRepository repo = new StudentRepository(); // tightly coupled
}
```

🔹 With Autowiring:

java

CopyEdit

```
@Component
public class StudentService {

   @Autowired
   private StudentRepository repo;
}
```

✅ Now Spring injects StudentRepository automatically. You don't need new.

✅ **Ways to Autowire in Spring**

Type	Example
Field injection	@Autowired private A a;
Setter injection	@Autowired public void setA(A a)
Constructor ✅	@Autowired public A(A a)

🔹 **Field Injection (not recommended for testing)**

java

CopyEdit

```
@Component
public class StudentService {
  @Autowired
  private StudentRepository repo;
}
```

🔹 **Setter Injection**

java

CopyEdit

```
@Component
public class StudentService {
  private StudentRepository repo;

  @Autowired
  public void setRepo(StudentRepository repo) {
    this.repo = repo;
```

```
    }
}
```

🔹 **Constructor Injection (Recommended ✅)**

java

CopyEdit

```
@Component
public class StudentService {
    private final StudentRepository repo;

    @Autowired
    public StudentService(StudentRepository repo) {
        this.repo = repo;
    }
}
```

✅ Helps with immutability
✅ Mandatory dependencies are clear
✅ Good for unit testing

🔹 **@Qualifier – When multiple beans exist**

If you have two beans of the same type, Spring doesn't know which one to inject. Use @Qualifier to specify the exact one.

java

CopyEdit

```
@Component("repo1")
public class StudentRepo1 implements StudentRepository { }

@Component("repo2")
```

```
public class StudentRepo2 implements StudentRepository { }


@Service
public class StudentService {


   @Autowired
   @Qualifier("repo2")
   private StudentRepository repo;
}
```

✅ **Real-Life Analogy**

Imagine Spring as a manager:

- @Autowired → Tells manager to find and assign the right employee (bean)
- @Qualifier("John") → Tells manager **which John** if there are two employees with the same name

---

✅ **Interview Questions**

1. What is @Autowired used for?
2. What is the difference between constructor, setter, and field injection?
3. When do you use @Qualifier?
4. Why is constructor injection preferred?

✅ Spring Core – Topic 16: Java-Based Configuration with @Configuration and @Bean

---

🔷 What Is Java-Based Configuration?

Instead of using applicationContext.xml, Spring allows you to configure your beans using **Java classes and annotations**.

✅ Why Use Java-Based Configuration?

- **Type-safe**
- **More readable**
- Easier to **refactor and debug**
- Integrated with **Java code** directly (no XML)

✅ Key Annotations

Annotation	Purpose
@Configuration	Declares a class as a **Spring configuration**
@Bean	Declares a **method that returns a Spring bean**

🔹 Example: @Configuration and @Bean

java

CopyEdit

```
@Configuration
public class AppConfig {

  @Bean
  public Student studentBean() {
    return new Student();
  }
```

```
    @Bean
    public StudentService studentServiceBean() {
        return new StudentService(studentBean());
    }
}
```

Spring will automatically register these beans in the application context.

✅ How to Enable Java Config

java

CopyEdit

```
AnnotationConfigApplicationContext context =
    new AnnotationConfigApplicationContext(AppConfig.class);

StudentService service = context.getBean(StudentService.class);
```

In Spring Boot, this is **enabled by default**, so no extra step needed.

✅ Comparison with XML

XML Config	Java Config
<bean id="x" class="..."/>	@Bean public X x() { return new X(); }
<context:component-scan ...>	@ComponentScan("package.name")

✅ Real-Life Analogy

Think of **Java-based config** like ordering via a well-designed web form (structured and validated) instead of using messy handwritten notes (XML).

✅ Best Practices

- Use @Configuration for app setup
- Use @Bean only when:
    - Third-party classes that cannot be annotated with @Component
    - You need full control over bean creation

✅ **Interview Questions**

1. What is @Configuration in Spring?
2. How is @Bean different from @Component?
3. Which is better: XML or Java-based configuration?
4. When should you use @Bean instead of @Component?

✅ Spring Core – Topic 17: Bean Scopes in Spring

🔷 What is a Bean Scope?

Bean scope defines **how many instances** of a bean Spring should create and **how long** the bean should live in the Spring container.

✅ Why Use Bean Scopes?

- To control **memory usage**
- To manage **state**
- To define **lifecycle** of beans as per need (singleton or per request/session)

✅ Types of Bean Scopes in Spring

Scope	Description	When to Use
singleton	One object per Spring container (default) ✅	Shared service, stateless logic
prototype	A new object each time it's requested	For non-shared, stateful beans
request	One bean per HTTP request (Web only)	Per request in web apps
session	One bean per HTTP session (Web only)	Per user session
application	One bean per application (ServletContext scope)	Shared app-wide object
websocket	One bean per WebSocket session	For WebSocket apps

🔹 1. Singleton (Default) ✅

java

CopyEdit

@Component

@Scope("singleton")

public class MySingletonBean { }

🟢 Only **one instance** created and shared
✅ Best for **services** and stateless logic

🔹 2. Prototype

java

CopyEdit

@Component

@Scope("prototype")

public class MyPrototypeBean { }

🟢 **New object** every time you getBean()
✅ Good for **stateful** or short-lived objects

🔹 3. Request (Web Apps Only)

java

CopyEdit

@Component

@Scope("request")

public class MyRequestBean { }

🟢 One bean instance per HTTP request
✅ Used in Spring MVC

🔹 4. Session (Web Apps Only)

java

CopyEdit

@Component

@Scope("session")

public class MySessionBean { }

🟢 One bean per HTTP session
✅ Used for **user-specific data**

✅ Bean Scope with @Bean

You can define scope for manually created beans too:

```
java
CopyEdit
@Configuration
public class AppConfig {

  @Bean
  @Scope("prototype")
  public Student student() {
    return new Student();
  }
}
```

✅ Real-Life Analogy

- **Singleton**: One car shared by everyone (e.g., a company car)
- **Prototype**: New car for every person
- **Request**: Cab booked for each customer
- **Session**: Netflix account for each user

✅ Interview Questions

1. What is the default scope of a Spring bean?
2. What is the difference between singleton and prototype?
3. When should you use prototype scope?
4. Can you use request/session scope in non-web applications?

Would you like the **PDF for To**

✅ Spring Core – Topic 18: Spring Bean Lifecycle and Lifecycle Methods

---

🔷 What is the Bean Lifecycle?

The **Bean Lifecycle** in Spring defines the **sequence of steps** a bean goes through from creation to destruction inside the **Spring container**.

✅ Why Is It Important?

- Helps perform **initialization or cleanup**
- Useful when working with **resources like DB connections**, file handlers, etc.
- Helps manage bean **state and memory**

🔹 Full Lifecycle Flow (Simplified):

1. **Object Instantiation**
2. **Dependency Injection**
3. **Bean Name Set**
4. **Aware Interfaces Called**
5. **Custom Init Method (@PostConstruct / init-method)**
6. **Bean Ready for Use**
7. **Custom Destroy Method (@PreDestroy / destroy-method)**

✅ Key Lifecycle Methods

Method Type	Use For	How to Define
@PostConstruct	Custom init logic	Annotate method after construction
@PreDestroy	Custom cleanup logic	Annotate method before destruction
init-method	Custom init (XML/Java config)	Defined in @Bean(init-method=...)
destroy-method	Custom cleanup (XML/Java)	Defined in @Bean(destroy-method=...)
afterPropertiesSet()	From InitializingBean	Runs after all properties set
destroy()	From DisposableBean	Runs during destruction

🔹 **Using @PostConstruct and @PreDestroy**

java

CopyEdit

```
@Component
public class StudentService {

  @PostConstruct
  public void init() {
    System.out.println("Bean is initialized");
  }

  @PreDestroy
  public void cleanup() {
    System.out.println("Bean is about to be destroyed");
  }
}
```

🔹 **Using InitializingBean and DisposableBean**

java

CopyEdit

```
@Component
public class MyBean implements InitializingBean, DisposableBean {

  @Override
  public void afterPropertiesSet() {
    System.out.println("Initializing logic here");
  }

  @Override
  public void destroy() {
    System.out.println("Cleanup logic here");
  }
}
```

✅ Used when you want **full control** via interface methods
✅ Avoid in modern apps – prefer annotations

🔹 **Java Config with init-method / destroy-method**

java

CopyEdit

```
@Configuration
public class AppConfig {

  @Bean(initMethod = "init", destroyMethod = "cleanup")
  public Student student() {
    return new Student();
```

```
    }
}
```

✅ Real-Life Analogy

Think of a bean like a **person joining and leaving a company**:

- @PostConstruct → Onboarding/training
- @PreDestroy → Exit/hand-over process
- afterPropertiesSet() → Setup after all data filled
- destroy() → Clear resources (badge, laptop)

✅ Interview Questions

1. What is the Spring Bean lifecycle?
2. What are @PostConstruct and @PreDestroy?
3. Difference between @PostConstruct and afterPropertiesSet()?
4. How to define custom init and destroy methods?
5. Which is preferred: annotation-based or interface-based lifecycle methods?

✅ Spring Core – Topic 19: Dependency Injection (DI) vs Inversion of Control (IoC)

🔷 What is Inversion of Control (IoC)?

IoC is a **design principle** where the control of object creation and management is transferred from the application code to the **framework** (Spring).

🔁 It "inverts" the usual flow of control.

🔹 Without IoC:

java

CopyEdit

StudentService service = new StudentService(new StudentRepository());

⛔ You are responsible for creating objects and wiring them.

🔹 With IoC (Spring):

java

CopyEdit

@Autowired

StudentService service;

✅ Spring manages the object and its dependencies — **you just use it**.

✅ What is Dependency Injection (DI)?

Dependency Injection is a technique through which Spring **provides required dependencies** (like objects or services) to a class **at runtime**.

💡 DI is the main way IoC is implemented in Spring.

✅ **Types of Dependency Injection in Spring**

Type	How It Works	Example
Constructor DI ✅	Dependencies passed via constructor	public A(B b)
Setter DI	Dependencies passed using setters	public void setB(B b)
Field DI ❌ (Not preferred)	Spring injects dependency directly into the field	@Autowired B b;

🔹 **Example of Constructor Injection (Recommended):**

java

CopyEdit

```
@Component
public class StudentService {
  private final StudentRepository repo;

  @Autowired
  public StudentService(StudentRepository repo) {
    this.repo = repo;
  }
}
```

✅ Helps with immutability
✅ Encourages better design
✅ Great for unit testing

🔹 **Setter Injection:**

java

CopyEdit

```
@Component
```

```
public class StudentService {
  private StudentRepository repo;

  @Autowired
  public void setRepo(StudentRepository repo) {
    this.repo = repo;
  }
}
```

🔹 **Field Injection (Not Recommended):**

java

CopyEdit

```
@Component
public class StudentService {
  @Autowired
  private StudentRepository repo;
}
```

⛔ Hard to test
⛔ Not recommended by Spring team for production code

✅ **Real-Life Analogy**

- IoC = Hiring manager assigns tasks to employees instead of each employee deciding their own job
- DI = Manager gives each employee their tools (dependencies) so they can do their job

---

✅ **Benefits of IoC & DI**

- ✅ **Loose Coupling** — Classes are independent

- ✅ **Reusable Code**
- ✅ **Easier Testing** — Mocks can be injected
- ✅ **Easier to maintain and extend**

✅ Interview Questions

1. What is IoC in Spring?
2. How is IoC related to DI?
3. What are the types of Dependency Injection in Spring?
4. Why is constructor injection preferred?
5. What are the drawbacks of field injection?

✅ Spring Core – Topic 20: BeanFactory vs ApplicationContext

🔷 What Are These?

Both BeanFactory and ApplicationContext are **Spring containers** that manage beans and provide **Dependency Injection**.

They are interfaces provided by Spring for managing the lifecycle and configuration of beans.

✅ 1. BeanFactory (Basic Container)

- Defined in: org.springframework.beans.factory.BeanFactory
- Lightweight container
- **Lazy loading** (beans are created when requested)
- Suitable for memory-constrained environments (like mobile)

🔹 Example:

java

CopyEdit

BeanFactory factory = new XmlBeanFactory(new ClassPathResource("beans.xml"));

MyBean obj = factory.getBean("myBean", MyBean.class);

✅ 2. ApplicationContext (Advanced Container)

- Defined in: org.springframework.context.ApplicationContext
- Superset of BeanFactory
- **Eager loading** (all singleton beans are created at startup)
- More features than BeanFactory:
    - Event publishing
    - Internationalization (i18n)
    - AOP integration
    - Web environment support

- Environment abstraction (for profiles)

🔹 **Example:**

java

CopyEdit

```
ApplicationContext context = new ClassPathXmlApplicationContext("beans.xml");
MyBean obj = context.getBean("myBean", MyBean.class);
```

🔍 **Key Differences Table**

Feature	BeanFactory	ApplicationContext ✅
Bean Instantiation	Lazy (when requested)	Eager (at startup)
Performance	Faster startup	Slightly slower (due to eager load)
Web/AOP/Event Support	❌ Limited	✅ Full support
Message Resource (i18n)	❌ Not available	✅ Available
Profile Support	❌ No	✅ Yes
Recommended for Modern Use	❌ Outdated	✅ Always use ApplicationContext

---

✅ **Which One Should You Use?**

Use Case	Container
Basic, legacy, low-memory apps	BeanFactory (rarely)
Modern Spring applications	ApplicationContext ✅

✅ **Real-Life Analogy**

- **BeanFactory** is like a **manual water tap** — opens only when needed
- **ApplicationContext** is like an **automated water system** — everything ready and flowing in advance

---

✅ **Interview Questions**

1. What is the difference between BeanFactory and ApplicationContext?
2. Which container loads beans lazily?
3. Why is ApplicationContext preferred in modern applications?
4. What extra features does ApplicationContext provide?

✅ Spring Core – Topic 21: Spring Expression Language (SpEL)

🔷 What is SpEL?

SpEL (Spring Expression Language) is a **powerful expression language** used in Spring to query and manipulate objects at runtime.

🔹 You can use SpEL to **access properties**, **call methods**, **perform arithmetic**, or even **evaluate logic**—all within Spring configurations or annotations.

✅ Why Use SpEL?

- Dynamically access bean values
- Perform logic within annotations or configurations
- Reduce boilerplate in property assignment

✅ Where Can You Use SpEL?

- In @Value annotations
- In Spring XML configurations
- Inside @Conditional, @Profile, etc.
- Inside Spring Security expressions

🔹 Basic Syntax

java

CopyEdit

@Value("#{expression}")

The #{} syntax is used for SpEL
The ${} syntax is for property placeholders

✅ Examples of SpEL

🔸 1. Accessing Bean Properties

java

CopyEdit

```
@Component
public class Student {
   private String name = "Viraj";
}

@Component
public class StudentService {
   @Value("#{student.name}")
   private String studentName;
}
```

🔸 **2. Calling Methods**

java

CopyEdit

```
@Value("#{T(java.lang.Math).random() * 100}")
private double randomValue;
```

✅ T() allows calling **static methods or classes**

🔸 **3. Performing Logical Operations**

java

CopyEdit

```
@Value("#{5 > 3}")
private boolean isTrue; // true
```

java

CopyEdit

```
@Value("#{student.age > 18 ? 'Adult' : 'Minor'}")
```

private String category;

🔸 **4. Accessing System Properties**

java

CopyEdit

@Value("#{systemProperties['user.name']}")

private String username;

🔸 **5. Combining with Properties File**

properties

CopyEdit

application.properties

course.name=Java Full Stack

java

CopyEdit

@Value("${course.name}")

private String course;

✅ Use ${} for **externalized config**,
✅ Use #{} for **dynamic SpEL logic**

✅ **Real-Life Analogy**

SpEL is like a **calculator** or **interpreter** inside Spring — you ask it to evaluate or fetch something **dynamically**, and it does it **at runtime**.

✅ **Benefits of SpEL**

- ✅ Supports complex expressions inside annotations
- ✅ Reduces need for boilerplate logic
- ✅ Powerful and flexible

- ✅ Supports accessing beans, properties, system environment, etc.

⚠️ Drawbacks

- ❌ Too much SpEL can reduce code readability
- ❌ Complex logic in annotations can be hard to debug
- ❌ Better to keep business logic in Java, not expressions

✅ Interview Questions

1. What is Spring Expression Language (SpEL)?
2. Where can you use SpEL in Spring?
3. Difference between ${} and #{}?
4. Can you call static methods using SpEL?
5. What are the pros and cons of using SpEL?

✅ Spring Core – Topic 22: Spring Profiles and Environment Management

---

🔷 What Are Spring Profiles?

Spring Profiles allow you to **define multiple configurations** for different environments (like dev, test, prod) and **switch between them easily**.

✅ Why Use Spring Profiles?

- 🧪 Use different beans/configs for different environments (dev vs prod)
- 🔐 Avoid hardcoding sensitive or environment-specific values
- 🛠️ Make your app **environment-aware** and **more flexible**

✅ How Spring Profiles Work

You annotate beans or config classes with:

java

CopyEdit

@Profile("dev")

Spring will **only activate** that bean/config if the profile is **enabled**.

🔹 Declaring a Profile on a Bean

java

CopyEdit

@Component

@Profile("dev")

public class DevDataSourceConfig implements DataSourceConfig {

  // Dev-specific bean logic

}

java

CopyEdit

```
@Component
@Profile("prod")
public class ProdDataSourceConfig implements DataSourceConfig {
   // Production-specific bean logic
}
```

🔹 **Activating a Profile**

✅ **Option 1: In application.properties**

properties

CopyEdit

```
spring.profiles.active=dev
```

✅ **Option 2: As a command-line argument**

bash

CopyEdit

```
java -jar app.jar --spring.profiles.active=prod
```

🔹 **Java-Based Profile Configuration**

java

CopyEdit

```
@Configuration
@Profile("dev")
public class DevConfig {
   @Bean
   public MyService devService() {
      return new MyService("Dev Mode");
   }
}
```

🔹 **Using Multiple Profiles**

properties

CopyEdit

spring.profiles.active=dev,test

✅ **What Is Environment Abstraction?**

Spring's Environment interface lets you programmatically access:

- Active Profiles
- Property values
- System environment variables

🔹 **Example:**

java

CopyEdit

@Autowired

Environment env;

public void checkProfile() {

String[] profiles = env.getActiveProfiles();

String dbUrl = env.getProperty("database.url");

}

✅ **Real-Life Analogy**

Think of Spring Profiles like **modes** in a phone:

- **Silent Mode (Profile: silent)**
- **Normal Mode (Profile: normal)**
- **Airplane Mode (Profile: airplane)**

Each mode activates a different set of behaviors/settings.

✅ Benefits of Spring Profiles

- ✅ Clean separation of environment-specific logic
- ✅ Avoids duplicate code
- ✅ Easy to switch configs during deployment
- ✅ Secure and modular

⚠️ Drawbacks

- ❌ Mismanagement of profile names can cause incorrect config loading
- ❌ Must remember to activate the correct profile during deployment
- ❌ Profiles can increase configuration complexity in large apps

---

✅ Interview Questions

1. What are Spring Profiles?
2. How do you activate a profile in Spring Boot?
3. What is the difference between @Profile and @Conditional?
4. Can you have multiple active profiles?
5. What is the Environment abstraction in Spring?

✅ Spring Core – Topic 23: Externalized Configuration with Properties and YAML

🔷 What is Externalized Configuration?

In Spring, **externalized configuration** means storing your application settings **outside the Java code**, typically in:

- application.properties
- application.yml

This makes your application **more flexible and environment-friendly** — change config without touching the code!

✅ Why Use Externalized Config?

- 🛠️ Make changes without recompiling
- 🔐 Keep sensitive data like DB passwords outside the code
- 📦 Easy to manage configs per environment (dev/test/prod)

✅ Common Configuration Files

File Type	Description
application.properties	Key-Value format (default)
application.yml	YAML format (cleaner and hierarchical)

✅ 1. Using application.properties

properties

CopyEdit

application.properties

server.port=8081

spring.datasource.url=jdbc:mysql://localhost:3306/mydb

spring.datasource.username=root

spring.datasource.password=admin123

✅ **2. Using application.yml**

yaml

CopyEdit

application.yml

server:

port: 8081

spring:

datasource:

url: jdbc:mysql://localhost:3306/mydb

username: root

password: admin123

✅ **3. Accessing Values in Code using @Value**

java

CopyEdit

@Value("${spring.datasource.username}")

private String dbUsername;

✅ **4. Mapping Properties to Class using @ConfigurationProperties**

java

CopyEdit

@Component

@ConfigurationProperties(prefix = "spring.datasource")

```
public class DataSourceConfig {
   private String url;
   private String username;
   private String password;

   // getters & setters
}
```

You can now autowire this class and get config easily.

✅ 5. Profile-Specific Configuration

You can have:

- application-dev.properties
- application-prod.yml

Spring will automatically load the one matching the active profile.

properties

CopyEdit

spring.profiles.active=dev

✅ Precedence Order (Which config is used first?)

1. Command line arguments
2. application.properties / application.yml
3. OS environment variables
4. Default values in code

✅ Real-Life Analogy

Think of it like customizing your home using a **remote configuration panel**. You don't need to break the wall (edit the code) — just change the settings in the control panel.

✅ Benefits

- ✅ Keeps code clean and flexible
- ✅ Easily supports different environments
- ✅ Easy config updates without rebuilding
- ✅ Sensitive info can be encrypted

⚠️ Drawbacks

- ❌ Too many config files can get confusing
- ❌ Forgetting to activate the right profile may load wrong settings
- ❌ Errors in YAML (indentation) can be tricky to debug

✅ Interview Questions

1. What is externalized configuration in Spring Boot?
2. How do application.properties and application.yml differ?
3. What is the use of @ConfigurationProperties?
4. How do you manage config for different environments?
5. What is the order of precedence for property sources?

✅ Spring Core – Topic 24: Summary & Quick Revision of Spring Core

---

This topic will help you **revise all core concepts** of the Spring Framework before jumping into advanced modules like Spring MVC or Spring Boot.

✅ 🌱 What Is Spring Framework?

Spring is a **lightweight**, **loosely coupled**, and **modular** framework used to build **Java enterprise applications**.

Its main goal is to **simplify development** using techniques like **Dependency Injection (DI)** and **Aspect-Oriented Programming (AOP)**.

✅ 🌼 Key Concepts in Spring Core (Quick Recap)

Concept	Purpose
IoC (Inversion of Control)	Delegates object creation to the Spring container
DI (Dependency Injection)	Injects dependencies automatically instead of manually creating them
Spring Beans	Objects managed by the Spring container
Bean Scopes	Defines bean lifecycle (singleton, prototype, etc.)
Bean Lifecycle	From creation → init → destruction
@Component/@Service...	Used to declare Spring beans automatically
@Autowired / @Qualifier	Used for automatic injection of dependencies
Configuration (@Config)	For declaring beans via Java class instead of XML
BeanFactory vs AppContext	Two containers: basic vs full-featured
SpEL	Expression language to inject dynamic values
Profiles	Run environment-specific configurations
application.properties/yml	Externalize configuration for flexibility

---

✅ 🧠 Annotations You Must Know

Annotation	Usage
@Component	Generic bean class
@Service	Business logic bean
@Repository	DAO layer bean
@Controller	Web controller
@Configuration	Configuration class for Java-based config
@Bean	Declare beans manually
@Autowired	Inject dependencies
@Qualifier	Resolve conflicts in injection
@Value	Inject values from properties
@Profile	Activate beans for specific environments

✅ Real-Life Flow: How Spring Core Works

1. Spring container starts
2. Loads bean definitions (from annotations or XML)
3. Instantiates beans based on scope
4. Injects dependencies into beans
5. Calls lifecycle methods (init/destroy)
6. Application is ready to use

✅ Benefits of Using Spring Core

- ✅ Loose coupling (via DI)
- ✅ Modular and scalable
- ✅ Test-friendly
- ✅ Saves development time

- ✅ Easy to maintain and reuse components

⚠️ Common Mistakes to Avoid

- ❌ Not managing scope properly (e.g., using singleton where prototype is needed)
- ❌ Confusing @Component vs @Bean
- ❌ Overusing XML (Java/annotation config is preferred now)
- ❌ Ignoring @Qualifier when multiple beans are present

✅ Interview Revision Questions

1. What is the difference between IoC and DI?
2. How are beans managed in Spring?
3. What is the purpose of @Component vs @Bean?
4. Difference between BeanFactory and ApplicationContext?
5. How does Spring handle different environments?

✅ What's Next?

Now that you've mastered **Spring Core**, the next logical steps are:

Next Module	What You'll Learn
Spring MVC	Building web apps with controllers, views
Spring Boot	Rapid development using auto-configuration
Spring Data JPA	Connecting to databases with ease
Spring Security	Adding login/authentication to your app