Spring Core

Spring Core – Introduction

---

1. Introduction to Spring Core

What is Spring Framework?

Spring is a powerful, lightweight, and open-source **Java framework** designed to simplify the development of **enterprise-level applications**.
It provides **comprehensive infrastructure support** for developing **Java applications** and promotes **good programming practices** like loose coupling and testability.

 Developed by: **Rod Johnson** in 2003
 Core Concept: **Inversion of Control (IoC)** and **Dependency Injection (DI)**

Why Use Spring?

Traditional Java EE (J2EE) applications are:

- Heavyweight
- Complex to configure (e.g., EJB)
- Hard to test (tightly coupled classes)

Spring overcomes these issues by:

- Reducing boilerplate code
- Promoting **POJO-based development**
- Making applications more **modular, testable, and maintainable**

Features of Spring Core

Feature	Description
Inversion of Control (IoC)	Delegates object creation to the container
Dependency Injection (DI)	Injects object dependencies at runtime
Lightweight Container	Spring uses POJOs instead of EJBs
Testability	Easily test units using mocks and no server required
Aspect-Oriented Programming (AOP)	Separates cross-cutting concerns
Transaction Management	Declarative and programmatic transaction support
Modular Architecture	You can use what you need (Core, MVC, JDBC, etc.)

Benefits of Using Spring

Benefit	Explanation
Lightweight	No heavy server needed, runs with minimal resources
Modular	Use only required modules (Core, ORM, AOP, etc.)
Loosely Coupled	Classes are not tightly dependent on each other
Easy Testing	Promotes mock testing with POJO
Configuration Flexibility	XML, Annotation, Java-based
Integration Ready	Works well with Hibernate, JPA, Struts, etc.

Role of IoC (Inversion of Control) and DI (Dependency Injection)

□ Inversion of Control (IoC):

- **Definition**: A design principle where control of object creation and wiring is transferred from the program to the container.
- Spring container manages the life cycle of objects (beans).

□ Dependency Injection (DI):

- **Definition**: A design pattern where dependencies (objects) are injected by the container at runtime rather than the class creating them.
- Promotes **loose coupling** and better **unit testing**.

Types of DI in Spring:

1. Constructor Injection
2. Setter Injection
3. Field Injection (via annotations)

Comparison: Spring vs Traditional Java EE

Aspect	Java EE (Traditional)	Spring Framework
Component Model	EJB (Heavy, complex)	POJO (Simple)
Configuration	XML or annotations	XML, Annotations, or Java Config
Testability	Difficult, needs container	Easy (No container required)
Coupling	Tightly Coupled	Loosely Coupled
Deployment	Requires app server	Can run standalone

Spring Core – List of Core Spring Modules

Spring Framework is composed of multiple modules that are **loosely coupled** but **integrated**, allowing developers to use only the parts they need.

---

1. Spring Core

What is it?

The **fundamental module** of the Spring Framework providing the **Inversion of Control (IoC)** and **Dependency Injection (DI)**.

Key Responsibilities:

- Manages bean creation and wiring

- Supports Constructor & Setter Injection

---

2. Spring Beans

What is it?

Provides **configuration and lifecycle management** of application objects (beans).

Key Responsibilities:

- Defines how beans are created, initialized, and destroyed
- Works with XML or Annotation-based configuration

---

3. Spring Context

What is it?

Built on top of the Core and Beans modules, this module provides **application context** and **enterprise-level services**.

Key Responsibilities:

- Event propagation
- Internationalization (i18n)
- Resource loading
- Bean factory extensions

---

4. Spring Expression Language (SpEL)

What is it?

A **powerful expression language** used for **value injection** into beans dynamically.

Key Capabilities:

- Accessing properties, calling methods

- Conditional expressions
- Collections, arrays, maps access

Example:

```
@Value("#{2 * T(Math).PI}")
private double circumference;
```

---

5. Spring AOP (Aspect-Oriented Programming)

What is it?

Enables **modularization of cross-cutting concerns** like logging, security, transactions.

Key Concepts:

- Aspect
- Join Point
- Advice (Before, After, Around)
- Pointcut

---

6. Spring JDBC

What is it?

Simplifies **raw JDBC** boilerplate by offering a cleaner and consistent API.

Key Features:

- Exception handling with DataAccessException
- JdbcTemplate class
- Reduces try-catch-finally verbosity

---

7. Spring ORM

What is it?

Provides integration with **popular ORM frameworks** like Hibernate, JPA, iBatis.

Key Features:

- Unified data access
- Simplified configuration
- Transaction management abstraction

---

8. Spring Transaction

What is it?

Supports **declarative and programmatic transaction management**.

Key Features:

- Abstracts underlying transaction API (JDBC, Hibernate, JPA)
- Annotations like @Transactional
- PlatformTransactionManager

---

9. Spring Web

What is it?

Provides **basic web integration** using the Servlet API.

Use Cases:

- Multipart file upload
- WebApplicationContext
- Basic request handling and filter integration

---

10. Spring Web MVC

What is it?

A **full-featured MVC framework** for building web apps and RESTful services.

Key Components:

- DispatcherServlet
- Controllers (@Controller, @RestController)
- ViewResolvers
- ModelAndView

---

11. Spring Security

What is it?

Provides **authentication, authorization, and access-control** features.

Features:

- Form-based login
- Role-based access control
- LDAP/Database authentication
- CSRF protection
- JWT integration (for REST APIs)

---

12. Spring Test

What is it?

Supports **unit testing and integration testing** of Spring applications.

Features:

- Works with JUnit/TestNG
- @SpringBootTest, @ContextConfiguration
- MockMvc for web-layer testing

Loosely Coupled Architecture in Spring

What is Coupling in Software Design?

Coupling refers to the **degree of dependency between software components or classes**.

- When one class relies heavily on another class's implementation, they are said to be **tightly coupled**.
- If a class depends only on the behavior (interface) and not on the implementation, it is **loosely coupled**.

Tight Coupling vs Loose Coupling

Aspect	Tight Coupling	Loose Coupling
Dependency	Classes directly depend on concrete classes	Classes depend on interfaces or abstractions
Flexibility	Low – hard to change implementation	High – easily replace components
Testing	Difficult – classes are hard to isolate	Easy – mock or swap dependencies
Reusability	Low – tightly bound behavior	High – interchangeable modules
Maintenance	Hard – changes ripple through system	Easy – well-encapsulated and isolated logic

---

Example – Tight Coupling

```
class Engine {
    public void start() {
        System.out.println("Engine started");
    }
}

class Car {
    Engine engine = new Engine(); // Tight coupling

    public void drive() {
        engine.start();
        System.out.println("Car is moving");
    }
}
```

Problems:

- If you want to use a PetrolEngine instead of Engine, you must **change the Car class**.
- Not flexible, hard to unit test.

Example – Loose Coupling using Interface (Spring Style)

```
interface Engine {
    void start();
}

class DieselEngine implements Engine {
    public void start() {
        System.out.println("Diesel engine started");
    }
}

class Car {
    private Engine engine;

    // Constructor Injection
    public Car(Engine engine) {
        this.engine = engine;
    }

    public void drive() {
        engine.start();
        System.out.println("Car is moving");
    }
}
```

Now, you can easily inject different types of engines:

```
Engine engine = new DieselEngine();
Car car = new Car(engine);
car.drive();
```

How Spring Promotes Loose Coupling

Spring encourages **Loose Coupling** through:

1. Interfaces and Abstraction

- Components depend on **interfaces**, not concrete classes.
- Promotes **Open-Closed Principle** from SOLID.

2. Dependency Injection (DI)

- Spring **injects dependencies** via:
  - Constructor
  - Setter
  - Field (via @Autowired)

```
<!-- XML Config -->
<bean id="engine" class="com.example.DieselEngine"/>
<bean id="car" class="com.example.Car">
   <constructor-arg ref="engine"/>
</bean>
```

3. Configuration Over Instantiation

- Object creation is done by the **Spring Container**, not by your classes.
- Business logic is **decoupled** from object lifecycle and wiring.

Benefits of Loose Coupling

Benefit	Description

Benefit	Description
Easy Testing	Mocks or stubs can replace real components
Better Flexibility	Easily switch implementations
Simpler Maintenance	Changing one component doesn't break others
Reusable Components	Classes work independently of specific implementations
Modular Design	Components are isolated and composable

Dependency Injection (DI) in Spring Core

What is Dependency Injection (DI)?

Dependency Injection (DI) is a **design pattern** used to achieve **Inversion of Control (IoC)** between classes and their dependencies.
Instead of a class creating its dependencies, they are **provided (injected)** by an external source (like Spring Container).

Role of DI in Spring Core

In Spring, DI is **the foundation** of the framework.
It allows the **Spring IoC container** to manage and inject dependencies between objects (beans) automatically, based on configuration (XML, Annotations, or Java-based).

Traditional (Manual) Dependency Creation

```
class Service {
   Repository repo = new Repository(); // Manual creation
}
```

Problem:

- Tightly coupled
- Difficult to test/mock

- Hard to change implementation

With Spring DI

```
class Service {
   private Repository repo;

   public Service(Repository repo) { // Injected via constructor
       this.repo = repo;
   }
}
```

Advantages:

- Loose coupling
- Easily replaceable and testable components
- Promotes clean architecture (like MVC, layered)

Types of Dependency Injection in Spring

---

1. Constructor Injection

Dependency is provided through class constructor

```
@Component
class Student {
   private Book book;

   @Autowired
   public Student(Book book) {
       this.book = book;
   }
}
```

Benefits:

- Makes object immutable
- Preferred for **mandatory dependencies**
- Clearer and cleaner design

2.Setter Injection

Dependency is set through a setter method

```
@Component
class Student {
    private Book book;

    @Autowired
    public void setBook(Book book) {
        this.book = book;
    }
}
```

Benefits:

- Flexible
- Ideal for **optional dependencies**

3.Field Injection (@Autowired)

Dependency is injected directly into the field

```
@Component
class Student {

    @Autowired
    private Book book;

}
```

Benefits:

- Less code (no setters/constructors needed)
- Quick for small projects

Caveat:

- Harder to unit test (requires reflection)
- Not recommended for large-scale projects

XML-based Injection (Alternative to Annotations)

Constructor:

```
<bean id="student" class="com.example.Student">
   <constructor-arg ref="book"/>
</bean>
```

Setter:

```
<bean id="student" class="com.example.Student">
   <property name="book" ref="book"/>
</bean>
```

Benefits of Using DI in Spring

Benefit	Description
Reusability	Components can be reused with different dependencies
Testability	Easy to inject mocks for unit testing
Reduced Boilerplate	No need for manual object creation or wiring
Maintainability	Easy to update and configure without changing business logic
Separation of Concerns	Business logic is separated from object management

What is IoC Container in Spring?

Definition

IoC (Inversion of Control) Container in Spring is the **core component** that is responsible for:

- Creating objects (beans)
- Injecting dependencies
- Managing the **lifecycle** of beans
- Configuring objects based on metadata (XML, annotations, or Java config)

In simpler terms: Spring’s IoC container **controls the flow of object creation**, not your application code.

Why "Inversion of Control"?

Traditionally, the developer writes code to **create and manage object dependencies**.

With IoC:

- This control is **inverted** — the **container does it for you**.
- Your classes only define *what* they need, not *how* to get it.

What Does the IoC Container Do?

Function	Description
Object Creation	Instantiates beans as defined in config
Dependency Injection	Injects dependencies into beans
Lifecycle Management	Handles init/destroy callbacks
Configuration Handling	Reads metadata (XML/Annotation/Java Config)

Types of IoC Containers in Spring

Spring provides **two main types** of containers:

Container Type	Interface	Key Features
BeanFactory	org.springframework.beans.factory.BeanFactory	Lightweight, lazy-loading container
ApplicationContext	org.springframework.context.ApplicationContext	Full-featured, eager-loading container

---

1. BeanFactory

- Basic container
- Loads beans **on-demand (lazy)** — good for memory-constrained environments
- Used mostly in early Spring versions or low-resource applications

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

---

2. ApplicationContext (Preferred)

- Superset of BeanFactory
- Loads all beans **at startup (eager)**
- Supports:
  - Internationalization (i18n)
  - Application events
  - BeanPostProcessor, BeanFactoryPostProcessor

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

Example: Using IoC with XML Config

beans.xml

```
<bean id="book" class="com.example.Book"/>
<bean id="student" class="com.example.Student">
   <constructor-arg ref="book"/>
</bean>
```

Main.java

```
ApplicationContext context = new
ClassPathXmlApplicationContext("beans.xml");
Student student = context.getBean("student", Student.class);
student.read();
```

Benefits of Using IoC Container

Benefit	Description
Loose Coupling	Objects don't create their own dependencies
Easy Testing	Mock dependencies can be injected
Central Configuration	All beans managed in one place
Lifecycle Management	Init, destroy hooks automatically handled
Modular Code	Promotes separation of concerns

What is a Spring Bean?

Definition

A **Spring Bean** is simply a **Java object that is managed by the Spring IoC container**.

When you define an object in Spring’s configuration (XML, annotations, or Java-based), Spring creates and manages it — **that object becomes a Spring Bean**.

Key Characteristics of a Spring Bean

Feature	Description
Managed by Container	Instantiated, wired, and destroyed by Spring
Configured via Metadata	XML, annotations (@Component, @Bean, etc.), or Java config
Supports Dependency Injection	Other beans can be injected into it
Lifecycle Handled by Spring	Init and destroy methods supported
Singleton by Default	But can be configured as prototype or others

---

How to Declare a Bean

---

1. XML Configuration

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

---

2. Annotation-Based (@Component)

```
@Component
public class Student { }
```

With component scanning enabled:

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

---

3. Java-based Configuration (@Bean)

```
@Configuration
public class AppConfig {

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

Spring Bean Lifecycle (Overview)

1. **Instantiation** – Bean object is created
2. **Populate Properties** – Dependencies injected
3. **BeanNameAware / BeanFactoryAware** – Aware interfaces called
4. **Pre-initialization** – BeanPostProcessor (before)
5. **Initialization** – @PostConstruct or custom init method
6. **Post-initialization** – BeanPostProcessor (after)
7. **Destruction** – @PreDestroy or custom destroy method (on container shutdown)

Example: Defining and Using a Bean

```
@Component
public class Book {
    public void read() {
        System.out.println("Reading a book...");
    }
}

@Component
public class Student {
    @Autowired
    Book book;

    public void study() {
        book.read();
    }
}
```

Main App:

```
ApplicationContext context = new
AnnotationConfigApplicationContext(AppConfig.class);
Student student = context.getBean(Student.class);
```

student.study();

Why Are Beans Important?

Benefit	Explanation
Central to Spring's IoC	Beans are the fundamental building blocks
Supports Loose Coupling	DI makes beans modular and testable
Reusable & Configurable	Easily reused and configured
Testable	Beans can be mocked or injected in tests

Spring IoC Containers – BeanFactory vs ApplicationContext

What is an IoC Container?

The **IoC (Inversion of Control) Container** is the **core of the Spring Framework**. It is responsible for:

- **Creating objects (beans)**
- **Injecting dependencies**
- **Managing lifecycle**
- **Handling configurations**

Spring provides **two main types** of containers:

1. **BeanFactory** – The basic container
2. **ApplicationContext** – The advanced and preferred container

---

1. BeanFactory

Overview:

- Defined by the interface: org.springframework.beans.factory.BeanFactory
- It is the **simplest** container, introduced in early Spring versions.
- Follows **lazy loading**: beans are created only when getBean() is called.

Usage Example:

```
Resource resource = new ClassPathResource("beans.xml");
BeanFactory factory = new XmlBeanFactory(resource);
MyBean bean = factory.getBean("myBean", MyBean.class);
```

Limitations:

- No built-in support for:
  - Application events
  - Internationalization (i18n)
  - Annotation-based configuration
- Not suitable for most modern applications

---

2. ApplicationContext

Overview:

- Sub-interface of BeanFactory
  → org.springframework.context.ApplicationContext
- More **feature-rich** and commonly used in real-world Spring projects.
- Follows **eager loading**: beans are created at the time of container initialization.

Features:

- All features of BeanFactory +
- Internationalization support (i18n)
- Event propagation (ApplicationEvent)
- Bean lifecycle callbacks (BeanPostProcessor, BeanFactoryPostProcessor)
- Annotation support (@Autowired, @Component, etc.)

Usage Example:

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

Key Differences: BeanFactory vs ApplicationContext

Feature	BeanFactory	ApplicationContext
Interface	BeanFactory	ApplicationContext (extends BeanFactory)
Loading Type	Lazy – loads bean when requested	Eager – loads beans at startup
Event Handling	Not supported	Supported (ApplicationEvent)
i18n Support	No	Yes (MessageSource)
AOP/Annotations	Limited	Full support
Bean Lifecycle Hooks	Not all supported	Supports BeanPostProcessor, etc.
Usage Scope	Lightweight/simple apps, testing only	Production-ready, web apps

---

Which One Should You Use?

- **Use ApplicationContext** in almost all Spring-based applications.
- BeanFactory is only used in rare cases where memory or startup time is a concern (e.g., IoT or mobile devices).

Difference Between BeanFactory and ApplicationContext in Spring

Feature / Aspect	BeanFactory	ApplicationContext
Definition	The **basic** Spring container that provides **lazy** initialization of beans.	A **more advanced** container that builds on BeanFactory, adding more features like event propagation, AOP integration, and internationalization.
Initialization	Beans are created **lazily** (on first request).	Beans are created **eagerly** (at startup by default).
Interface Type	Part of the org.springframework.beans.factory package.	Part of the org.springframework.context package.

Feature / Aspect	BeanFactory	ApplicationContext
Features Provided	Only basic **dependency injection** support.	Provides full **enterprise features**: event handling, message source, annotation support, etc.
Bean Post Processors & Aware Interfaces	Not automatically supported (must be manually registered).	Automatically detects and registers BeanPostProcessor and BeanFactoryPostProcessor.
Event Publishing	Not supported	Supported
Internationalization (i18n)	Not supported	Supported via MessageSource
Preferred Usage	Suitable for **lightweight applications** or **memory-constrained devices**.	Recommended for **most enterprise** or **web applications**.
Common Implementations	XmlBeanFactory (deprecated in Spring 3.1)	ClassPathXmlApplicationContext, FileSystemXmlApplicationContext, AnnotationConfigApplicationContext, WebApplicationContext

---

In Simple Terms:

- **BeanFactory** = Light, fast, manual, basic DI container
- **ApplicationContext** = Heavy, powerful, full-featured container for most apps

Setter Injection vs Constructor Injection in Spring XML

1.Constructor Injection

- Dependencies are passed into the bean using a **constructor** at the time of bean creation.
- Uses <constructor-arg> tag.

Example:

```
<bean id="employee" class="com.example.Employee">
   <constructor-arg value="101"/>
```

```
    <constructor-arg value="Prathamesh"/>
</bean>
```

Corresponding Java Class:

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

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

---

2. Setter Injection

- Dependencies are injected using **setter methods after** bean instantiation.
- Uses <property> tag.

Example:

```
<bean id="employee" class="com.example.Employee">
    <property name="id" value="101"/>
    <property name="name" value="Prathamesh"/>
</bean>
```

□ **Corresponding Java Class:**

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

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

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

}

Key Differences

Feature	Constructor Injection	Setter Injection
Method Used	Constructor	Setter methods
Injection Time	At bean instantiation	After bean instantiation
Flexibility	Less (must pass all values)	More (can set selected values)
Unit Testing	Harder to mock	Easier to mock
Circular Dependency	May fail (if both beans depend on each other via constructor)	Works better

---

When to Use What?

Use Setter Injection When...	Use Constructor Injection When...
Partial or optional injection needed	All mandatory dependencies required
You want more control / flexibility	You want immutability
Circular dependencies possible	Small number of simple dependencies

Spring Bean Scopes

In Spring, a **bean scope** defines the **lifecycle and visibility** of a bean—i.e., how many instances of the bean are created and how they are shared across the application.

---

1. singleton (Default Scope)

- Only **one instance per Spring IoC container**.
- Same object returned every time the bean is requested.
- Default in XML and Annotation configs.

Example:

```
<bean id="myBean" class="com.example.MyBean" scope="singleton" />
```

Java Annotation:

```
@Component
@Scope("singleton")
```

---

2. prototype

- A **new bean instance** is created every time it is requested.
- Spring container creates and **returns new object**, but **does not manage** its complete lifecycle (e.g., no destroy method).

Example:

```
<bean id="myBean" class="com.example.MyBean" scope="prototype" />
```

Java Annotation:

```
@Component
@Scope("prototype")
```

---

3. request (Web-aware scope)

- One bean instance **per HTTP request**.
- Only valid in **Spring Web (Spring MVC)** applications.

Annotation Example:

```
@Component
@Scope(value = WebApplicationContext.SCOPE_REQUEST, proxyMode =
ScopedProxyMode.TARGET_CLASS)
```

---

4. session (Web-aware scope)

- One bean instance **per HTTP session**.
- Retained across multiple requests from the same user.

Annotation Example:

```
@Component
@Scope(value = WebApplicationContext.SCOPE_SESSION, proxyMode =
ScopedProxyMode.TARGET_CLASS)
```

---

5. application

- One bean instance for the **entire ServletContext** (web app-wide).

Annotation Example:

```
@Component
@Scope(value = WebApplicationContext.SCOPE_APPLICATION, proxyMode =
ScopedProxyMode.TARGET_CLASS)
```

---

6. websocket

- One bean instance per **WebSocket session**.

Available only in Spring WebSocket environments.

Types of Bean Scopes in Spring Core

Spring defines **5 main types of bean scopes**, which determine the lifecycle and visibility of a bean inside the application.

---

1. Singleton (Default Scope)

- **Definition**: Only **one instance** of the bean is created for the **entire Spring container**.
- **Usage**: Ideal for **stateless beans** (services, configs).
- **Lifecycle**: Created at container startup, reused on every request.

□ **XML Configuration:**

```
<bean id="myBean" class="com.example.MyBean" scope="singleton" />
```

□ **Annotation:**

```
@Component
@Scope("singleton")
```

□ **Characteristics:**

- Shared across multiple components.
- Best for lightweight, thread-safe beans.

---

2. Prototype

- **Definition**: A **new instance** is created every time the bean is requested.
- **Usage**: Useful when bean state should not be shared (non-thread-safe or mutable objects).
- **Lifecycle**: Created on-demand; Spring does **not manage** destruction.

□ **XML Configuration:**

```
<bean id="myBean" class="com.example.MyBean" scope="prototype" />
```

□ **Annotation:**

```
@Component
@Scope("prototype")
```

Characteristics:

- Not suitable for dependency injection in singleton beans without special handling.
- You must manage cleanup manually.

---

3. Request (Web-only)

- **Definition**: One bean instance per **HTTP request**.
- **Usage**: Useful in web applications for **per-request data**, like form inputs.
- **Lifecycle**: Exists during a single HTTP request, destroyed afterward.

Annotation:

@Component
@Scope(value = WebApplicationContext.SCOPE_REQUEST, proxyMode = ScopedProxyMode.TARGET_CLASS)

Characteristics:

- Available only in Spring Web or Spring Boot web applications.
- Requires DispatcherServlet context.

---

4. Session

- **Definition**: One bean instance per **HTTP session**.
- **Usage**: Store **user-specific information** (e.g., login session, shopping cart).
- **Lifecycle**: Exists as long as the session is active.

Annotation:

@Component
@Scope(value = WebApplicationContext.SCOPE_SESSION, proxyMode = ScopedProxyMode.TARGET_CLASS)

Characteristics:

- Maintains user-specific state.
- Automatically destroyed when session ends.

---

5. WebSocket

- **Definition**: One bean instance per **WebSocket session**.
- **Usage**: For **real-time applications** with persistent socket connections.

- **Lifecycle**: Tied to WebSocket session duration.

Annotation:

```
@Component
@Scope(value = "websocket", proxyMode =
ScopedProxyMode.TARGET_CLASS)
```

Characteristics:

- Requires Spring WebSocket support.
- Great for chat apps, dashboards, etc.

Autowiring in Spring Core

What is Autowiring?

Autowiring is a feature in Spring that allows **automatic injection of dependencies** (beans) into other beans without explicitly specifying them in the XML configuration.

It reduces boilerplate code and simplifies bean wiring.

How It Works

Spring looks for **matching bean definitions** in the container based on **property name**, **type**, or **constructor arguments**, and automatically injects the dependency.

You can define autowiring in:

- **XML Configuration**
- **Annotations** (@Autowired, @Qualifier)

Modes of Autowiring

Spring supports **4 main autowiring modes** when using XML configuration:

---

1. no (Default)

- **Description**: Autowiring is **disabled**.
- **Manual** wiring required via ref attribute.

```
<bean id="student" class="com.example.Student" autowire="no">
    <property name="address" ref="address" />
</bean>
```

---

2. byName

- **Description**: Spring looks for a bean **with the same name as the property name**.
- **Match**: Bean ID must match the property name.

```
<bean id="student" class="com.example.Student" autowire="byName" />
<bean id="address" class="com.example.Address" />
```

If Student has a property address, Spring will inject the address bean.

---

3. byType

- **Description**: Spring injects a bean based on **data type (class/interface)** of the property.
- **Match**: Class type must be **unique** in the context.

```
<bean id="student" class="com.example.Student" autowire="byType" />
<bean id="address1" class="com.example.Address" />
```

If multiple beans of the same type exist, it will throw an error.

---

4. constructor

- **Description**: Spring matches beans to **constructor arguments** by type.
- Useful for **constructor-based DI**.

```
<bean id="student" class="com.example.Student" autowire="constructor" />
<bean id="address" class="com.example.Address" />
```

If Student constructor requires an Address, Spring will inject the address bean.

Example Class for Autowiring

```
public class Student {
    private Address address;

    public void setAddress(Address address) {
        this.address = address;
    }

    public Student(Address address) {
        this.address = address;
    }
}
```

Summary Table

Mode	Matching Based On	Limitation	Suitable For
no	Manual wiring	More boilerplate	Full control
byName	Property name	Bean ID must match property name	Simple projects
byType	Data type	Only 1 matching type allowed	Cleaner for services
constructor	Constructor type	Requires matching constructor args	Immutable beans

---

Note: In annotation-based config, autowiring is done using:

```
@Autowired
private Address address;
```

To resolve ambiguity with @Autowired, use @Qualifier("beanName").

Spring Bean Life Cycle

In the **Spring Framework**, the *Bean Life Cycle* represents the various **stages a Spring Bean goes through**, from **instantiation** to **destruction**, under the control of the Spring IoC container.

Bean Life Cycle Phases

1. **Instantiation**
    - Spring creates a bean instance using the constructor (default or parameterized).
    - At this stage, the object is just created — no properties are set yet.
2. **Populate Properties (Dependency Injection)**
    - Spring injects dependencies (DI) into the bean via **setter methods** or **constructor injection**.
3. **Set Bean Name**
    - If the bean implements BeanNameAware, the container calls:

      ```
      setBeanName(String name)
      ```

    - This gives the bean access to its ID defined in the configuration.
4. **Set Bean Factory**
    - If the bean implements BeanFactoryAware, the container calls:

      ```
      setBeanFactory(BeanFactory factory)
      ```

    - Allows access to the BeanFactory object.
5. **Set ApplicationContext**
    - If the bean implements ApplicationContextAware, the method is called:

setApplicationContext(ApplicationContext context)

6. **Pre-initialization**
   - Spring calls any BeanPostProcessor's postProcessBeforeInitialization() method (if defined).
   - Useful for modifying bean properties before initialization.
7. **Initialization**
   - Spring calls the **custom init method** defined via:
     - XML: init-method
     - Annotation: @PostConstruct
     - Or if the bean implements InitializingBean, the method:

       afterPropertiesSet()
8. **Post-initialization**
   - Spring calls postProcessAfterInitialization() of BeanPostProcessor.
9. **Ready to Use**
   - Bean is fully initialized and ready for use in the application.
10. **Destruction**
    - When the container shuts down:
      - If the bean implements DisposableBean, Spring calls:

        destroy()
      - Or calls a custom destroy-method or @PreDestroy annotated method.

Life Cycle with XML Configuration

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

```
Constructor Called
      ↓
Dependencies Injected
      ↓
Aware Interfaces Called (BeanNameAware, BeanFactoryAware)
      ↓
BeanPostProcessor (before init)
      ↓
Initialization (afterPropertiesSet/init-method)
      ↓
BeanPostProcessor (after init)
      ↓
Bean Ready for Use
      ↓
Destruction (destroy()/destroy-method)
```

BeanPostProcessor in Spring Core

What is BeanPostProcessor?

BeanPostProcessor is an interface provided by the **Spring Framework** that allows **custom modification of new bean instances** *before* and *after* their initialization.

It acts as an **interceptor** for Spring bean creation and is a key component in **bean customization**, **aspect-oriented programming (AOP)**, and **framework-level features** like @Autowired, @PostConstruct, etc.

Execution Flow

Constructor → set properties →
BeanPostProcessor.postProcessBeforeInitialization() →
init-method / afterPropertiesSet() →
BeanPostProcessor.postProcessAfterInitialization()

Interface Definition

```
public interface BeanPostProcessor {
    Object postProcessBeforeInitialization(Object bean, String beanName) throws 
BeansException;
    Object postProcessAfterInitialization(Object bean, String beanName) throws 
BeansException;
}
```

Purpose of Each Method

Method	Description
postProcessBeforeInitialization()	Called **before** the custom init method (init-method or @PostConstruct) is invoked. You can modify the bean here.
postProcessAfterInitialization()	Called **after** the bean is fully initialized. Often used for wrapping the bean (like with proxies).

Use Cases of BeanPostProcessor

1. **Custom logging or validation**
2. **Injecting additional dependencies manually**
3. **Wrapping beans with proxies (used internally by Spring AOP)**
4. **Implementing annotations like @Autowired, @PostConstruct**
5. **Framework-specific customizations**

Example: Custom BeanPostProcessor

```
public class MyBeanPostProcessor implements BeanPostProcessor {

    @Override
    public Object postProcessBeforeInitialization(Object bean, String beanName) 
throws BeansException {
        System.out.println("Before Initialization of: " + beanName);
        return bean; // You can return a proxy or modified bean here
    }
    @Override
    public Object postProcessAfterInitialization(Object bean, String beanName)
throws BeansException {
        System.out.println("After Initialization of: " + beanName);
        return bean; // Wrap or modify the bean
    }
}
```

Registering BeanPostProcessor (XML Configuration)

```
<bean class="com.example.MyBeanPostProcessor" />
<bean id="myBean" class="com.example.MyBean" />
```

Important Notes

- It applies to **every bean** in the application context.
- You can define **multiple BeanPostProcessor** beans — Spring will apply them in order.
- Return value of the postProcess...() methods can be a **proxy** or **modified object**.

Comparison: Init Method vs BeanPostProcessor

Feature	init-method / @PostConstruct	BeanPostProcessor
Scope	Specific to one bean	Applied to **all beans**
Purpose	Setup logic for that bean	Interception/modification mechanism
Common Usage	Set default values, init setup	Logging, proxying, dependency tweak

Custom Initialization and Destruction Methods in Spring Core

Purpose

Spring allows you to define **custom logic** that should run:

- **after** the bean is fully initialized (i.e., after dependency injection)
- **before** the bean is destroyed (i.e., when the container is shutting down)

This is useful for:

- Initializing resources (e.g., DB connections, socket listeners)
- Releasing resources (e.g., closing files, clearing caches)

---

1. Using init-method and destroy-method in XML

XML Configuration Example:

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

Java Bean:

```
public class MyBean {
    public void init() {
        System.out.println("Custom Init Method: Bean is initialized");
    }

    public void cleanup() {
        System.out.println("Custom Destroy Method: Bean is about to be destroyed");
    }
}
```

---

2. Using Annotations: @PostConstruct and @PreDestroy

Requires javax.annotation dependency (included in Spring Context)

Annotated Java Bean:

```
import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;

public class MyBean {

    @PostConstruct
    public void customInit() {
        System.out.println("Init with @PostConstruct");
    }

    @PreDestroy
    public void customDestroy() {
        System.out.println("Destroy with @PreDestroy");
    }
}
```

---

3. Using InitializingBean and DisposableBean Interfaces

These are **Spring-specific interfaces**.

```
import org.springframework.beans.factory.InitializingBean;
import org.springframework.beans.factory.DisposableBean;

public class MyBean implements InitializingBean, DisposableBean {

    @Override
    public void afterPropertiesSet() throws Exception {
        System.out.println("InitializingBean: afterPropertiesSet called");
    }

    @Override
    public void destroy() throws Exception {
        System.out.println("DisposableBean: destroy called");
    }
```

}

Comparison of All Methods

Method	Type	Used For	Flexibility	Preferred?
init-method, destroy-method	XML config	External lifecycle config	Medium	Good
@PostConstruct, @PreDestroy	Annotation	Internal to class	High	Preferred
InitializingBean, DisposableBean	Interface	Tight coupling	Low	Avoid unless needed

---

Important Notes

- Spring only calls destroy-method on **singleton scoped beans**.
- Always register a **shutdown hook** if using AnnotationConfigApplicationContext or GenericXmlApplicationContext:

  ```
  context.registerShutdownHook();
  ```

When are they triggered?

Lifecycle Phase	Method Used
After dependencies set	@PostConstruct, init-method, afterPropertiesSet()
Before bean destruction	@PreDestroy, destroy-method, destroy()

Java-based Configuration in Spring Core (No XML)

What is Java-based Configuration?

In Java-based configuration, we use **Java classes and annotations** to define Spring Beans and configure the application instead of using traditional **XML configuration files**.

This approach is:

- **Type-safe**
- **Easier to refactor**
- **More concise and modern**

□ Key Annotations

Annotation	Description
@Configuration	Marks a class as the source of bean definitions (like <beans> in XML)
@Bean	Declares a bean definition (like <bean> in XML)
@ComponentScan	Tells Spring to scan the package for components (@Component, @Service)
@Component	Marks a class as a Spring-managed component
@Autowired	Injects dependencies automatically

Example – Simple Java Configuration

1. Bean Class

```
public class HelloService {
    public void sayHello() {
        System.out.println("Hello from HelloService!");
    }
}
```

---

2. Configuration Class

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class AppConfig {
    @Bean
    public HelloService helloService() {
        return new HelloService();
    }
}
```

---

3. Main Class

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

public class MainApp {
    public static void main(String[] args) {
        ApplicationContext context = new
AnnotationConfigApplicationContext(AppConfig.class);
        HelloService service = context.getBean(HelloService.class);
        service.sayHello();
    }
}
```

With Component Scanning

1. Component Class

```
import org.springframework.stereotype.Component;

@Component
public class HelloService {
    public void sayHello() {
        System.out.println("Hello from Component Scanned HelloService!");
    }
}
```

---

2. Configuration Class with Scanning

```
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
```

```
@Configuration
@ComponentScan(basePackages = "com.example")  // specify your base package
public class AppConfig {
}
```

Benefits of Java-Based Configuration

- Completely avoids XML
- Refactor-friendly due to compile-time checking
- Cleaner and more expressive
- Works well with annotations like @Autowired
- Ideal for Spring Boot and modern Spring applications

Notes

- Java-based config can coexist with XML if needed.
- It's highly recommended in modern Spring development (especially Spring Boot).
- All Spring Boot apps internally use Java-based configuration.

Component Scanning & Stereotype Annotations in Spring

What is Component Scanning?

Component Scanning is a feature in Spring that allows automatic detection of beans from the classpath (i.e., classes annotated with stereotype annotations) and registers them in the Spring container.

Instead of manually declaring beans using @Bean or XML, Spring auto-detects and wires them when annotated properly.

Stereotype Annotations

These annotations indicate that a class is a **Spring-managed component**. All of these are specializations of @Component.

Annotation	Purpose	Typical Usage Layer
@Component	Generic component	Utility, general
@Service	Denotes a service component	Business logic layer
@Repository	Marks DAO (data access) class	Persistence layer
@Controller	Indicates a web controller (Spring MVC)	Web layer
@RestController	Shortcut for @Controller + @ResponseBody	RESTful API

- All of the above are registered automatically during component scanning.

How to Enable Component Scanning

Using Java Configuration:

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

Using XML Configuration:

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

Example

1. Component Class

```
import org.springframework.stereotype.Component;

@Component
public class MyComponent {
    public void doSomething() {
        System.out.println("Doing something...");
    }
}
```

2. Configuration Class

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

3. Main Class

```
public class MainApp {
    public static void main(String[] args) {
        ApplicationContext context = new
AnnotationConfigApplicationContext(AppConfig.class);
        MyComponent obj = context.getBean(MyComponent.class);
        obj.doSomething();
    }
}
```

Notes

- Stereotype annotations make Spring applications more declarative and readable.
- Use @ComponentScan to define which packages to scan.
- Spring will **not scan** or register any class without these annotations or manual configuration.
- You can limit scanning to specific packages for better performance.

Best Practices

- Use specific annotations (@Service, @Repository, etc.) instead of generic @Component for clarity.
- Keep package structure organized (e.g., controller, service, repository) for effective scanning.

- Use @Qualifier if multiple beans of the same type exist.

Property Injection from External Files in Spring

What is Property Injection?

Property Injection allows you to inject values like URLs, usernames, passwords, configuration constants, etc., **from external files** (e.g., .properties) into your Spring beans. This promotes **externalized configuration**, separating config data from code.

Step-by-Step Setup Using .properties File

---

1. Create application.properties (or any .properties file)

```
# config.properties
app.name=SpringDemoApp
app.version=1.0
```

---

2. Create Java Bean to Inject Properties

```
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

@Component
public class AppConfigReader {

   @Value("${app.name}")
   private String appName;

   @Value("${app.version}")
   private String appVersion;
```

```
    public void display() {
        System.out.println("App Name: " + appName);
        System.out.println("App Version: " + appVersion);
    }
}
```

---

3. Load Properties in Configuration

```
import org.springframework.context.annotation.*;
import
org.springframework.context.support.PropertySourcesPlaceholderConfigurer;

@Configuration
@ComponentScan(basePackages = "com.example")
@PropertySource("classpath:config.properties")
public class AppConfig {

    // Required to resolve ${...} in @Value
    @Bean
    public static PropertySourcesPlaceholderConfigurer propertyConfigurer() {
        return new PropertySourcesPlaceholderConfigurer();
    }
}
```

---

4. Main Class to Run the Application

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

public class MainApp {
    public static void main(String[] args) {
        ApplicationContext context = new
AnnotationConfigApplicationContext(AppConfig.class);
        AppConfigReader reader = context.getBean(AppConfigReader.class);
        reader.display();
    }
```

}

Notes

- @Value("${key}") injects the property value.
- @PropertySource specifies the path to the .properties file.
- PropertySourcesPlaceholderConfigurer bean is **mandatory** for @Value to work.
- The file must be in the **classpath** (src/main/resources in Maven/Gradle projects).

Benefits

- Clean separation of code and configuration.
- Easily change values without rebuilding code.
- Supports different environments (dev, prod, test).

Spring Profiles – Environment Specific Configuration

What are Spring Profiles?

Spring Profiles allow you to define **different configurations** for **different environments** (e.g., development, testing, production) and activate them **dynamically**.

Why Use Profiles?

- Avoid hardcoding values/configs for multiple environments.
- Load only the beans and configurations needed for the active environment.
- Make deployment environment-agnostic.

Structure Overview

You can define:

- **Beans per profile** using @Profile
- **Different property files** per profile (e.g., dev.properties, prod.properties)
- Activate profiles via:
  - Code
  - Properties file
  - Command line

Example 1: Profile-Based Bean Creation

1. Define Beans with @Profile

```
@Component
@Profile("dev")
public class DevDataSourceConfig {
    public DevDataSourceConfig() {
        System.out.println("Loaded DEV DataSource Bean");
    }
}

@Component
@Profile("prod")
public class ProdDataSourceConfig {
    public ProdDataSourceConfig() {
        System.out.println("Loaded PROD DataSource Bean");
    }
}
```

---

2. Configuration Class

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

---

3. Main Class with Profile Activation

```
public class MainApp {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext();
        context.getEnvironment().setActiveProfiles("dev"); // activate "dev"
        context.register(AppConfig.class);
        context.refresh();
    }
}
```

Example 2: Different Properties for Profiles

Files:

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

properties

```
# application-dev.properties
db.url=jdbc:mysql://localhost/devdb

# application-prod.properties
db.url=jdbc:mysql://localhost/proddb
```

Using in Bean

```
@Value("${db.url}")
private String dbUrl;
```

Activating Profiles

Method	Example
In Code	context.getEnvironment().setActiveProfiles("dev")
In application.properties	spring.profiles.active=dev

Method	Example
JVM Argument	-Dspring.profiles.active=dev
Environment Variable	SPRING_PROFILES_ACTIVE=dev

Benefits

- Clean separation between dev/test/prod config.
- Reduces bugs due to wrong configs.
- Easy to manage and deploy in multiple environments.

Differences Between Spring Core and Spring Boot

Feature	Spring Core	Spring Boot
Definition	The **core module** of the Spring Framework providing features like **IoC** and **Dependency Injection**	A **framework built on top of Spring** that simplifies Spring application setup with **auto-configuration and starter dependencies**
Configuration	Requires **manual configuration** using XML or Java	Uses **auto-configuration**, reducing boilerplate
Dependencies	You need to manually manage all required dependencies	Uses **starter dependencies** (e.g., spring-boot-starter-web) to bundle needed libraries
Setup Time	Longer setup due to manual bean declaration and XML configurations	Much faster setup and development cycle
Main Components	BeanFactory, ApplicationContext, IoC, AOP, DI, etc.	Embedded server, starters, auto-configuration, Spring Boot CLI
Web Server	No embedded server by default; needs external setup (e.g., Tomcat)	Comes with **embedded servers** (Tomcat, Jetty, etc.)

Feature	Spring Core	Spring Boot
Testing Support	Manual testing setup	Integrated testing features (@SpringBootTest)
Main Focus	Provides fundamental building blocks for Spring applications	Provides **production-ready**, easy-to-deploy applications
Boilerplate Code	More boilerplate code to write	Minimal boilerplate; convention over configuration
Project Structure	Flexible structure, but no convention	Follows opinionated and standardized structure
Spring Initializr	Not available	Easily create projects using Spring Initializr
Runtime Efficiency	Efficient but depends on manual optimization	Optimized with sensible defaults and production-ready features

---

Use Cases

- **Spring Core** is ideal when:
  - You want **full control** over configuration.
  - You're working on **lightweight modules** or **learning core concepts**.
- **Spring Boot** is ideal when:
  - You want to build **REST APIs or microservices** quickly.
  - You prefer **rapid development** with **less setup**.