Palash Bajpai

Spring

Table of Contents

[**1. Introduction to Spring Framework** 2](#_Toc196758610)

[🔹 **How Spring solves EJB problems** 2](#_Toc196758611)

[🔹 **Spring Ecosystem** 3](#_Toc196758612)

[**2. Spring Core Concepts** 4](#_Toc196758613)

[🔹 **Inversion of Control** 4](#_Toc196758614)

[🔹 **Dependency Injection** 5](#_Toc196758615)

[🔹 **How IoC and DI really helps??** 6](#_Toc196758616)

[**3. Basic Spring terms** 8](#_Toc196758617)

[🔹 **Bean** 8](#_Toc196758618)

[🔹 **IoC container** 9](#_Toc196758619)

[🔹 **Autowiring** 11](#_Toc196758620)

[🔹 **Types of Dependency Injection** 12](#_Toc196758621)

[**4. XML-Based Configuration in Spring** 13](#_Toc196758622)

[🔹 **Scope of Bean** 14](#_Toc196758623)

[🔹 **Setter Injection** 15](#_Toc196758624)

[🔹 **Constructor Injection** 18](#_Toc196758625)

[🔹 **Working with Interfaces** 19](#_Toc196758626)

[🔹 **Autowiring** 20](#_Toc196758627)

# **1. Introduction to Spring Framework**

*Spring Framework is a lightweight, open-source Java framework for building enterprise-grade applications using dependency injection and aspect-oriented programming.*

Before Spring, Java developers primarily used Enterprise JavaBeans (EJB) for building enterprise applications. However, EJBs were:

* Heavyweight and complex
* Difficult to test (required containers like JBoss or WebLogic)
* Tightly coupled components (hard to manage dependencies)
* Required lots of boilerplate code

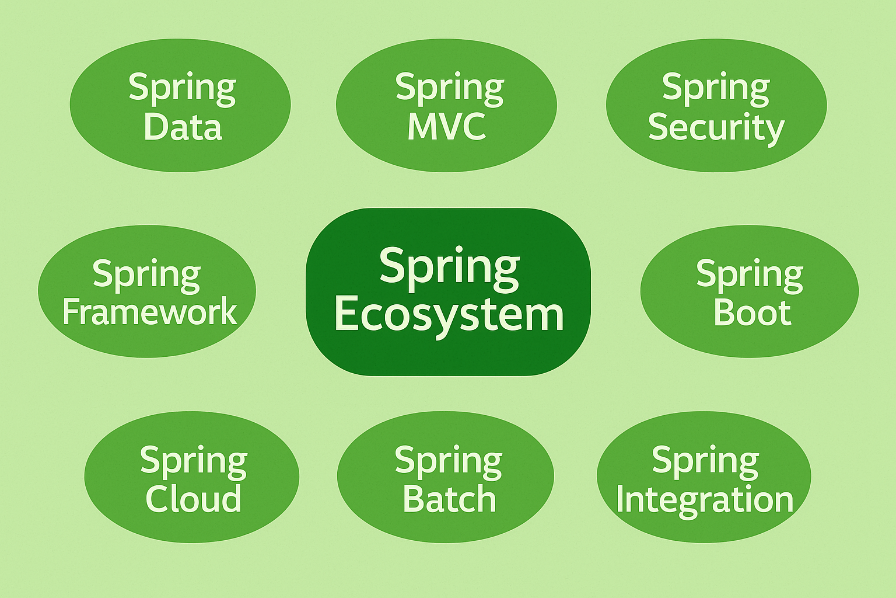
## 🔹 **How Spring solves EJB problems**

| **EJB Problem** | **Explanation (EJB)** | **Spring’s Solution** | **Explanation (Spring)** |
| --- | --- | --- | --- |
| **Heavyweight Containers** | Requires full-fledged application servers (e.g., JBoss, WebLogic) to run EJBs. | Uses lightweight containers (BeanFactory, ApplicationContext). | Spring can run in any simple Java environment (even a main method). |
| **Complex Configuration** | EJBs required verbose deployment descriptors (XML files) and JNDI lookups. | Spring uses annotations (@Component, @Autowired) and Java-based config. | Reduces boilerplate and improves readability and maintainability. |
| **Tight Coupling** | Business logic was tightly bound to the EJB container and interfaces. | Promotes loose coupling via Dependency Injection (DI) and interfaces. | Classes depend on abstractions, not implementations. |
| **Poor Testability** | EJBs couldn’t be unit tested easily without an EJB container. | Spring beans are POJOs — easy to mock, inject, and unit test. | Works with JUnit, Mockito, etc. |
| **Verbose Codebase** | Boilerplate for JNDI, transactions, remote/local interfaces. | Removes boilerplate using templates (e.g., JdbcTemplate, AOP for txns). | Code is concise and business-focused. |
| **Rigid Architecture** | Hard to plug in components or switch implementations. | Modular architecture allows easy swapping and extension of features. | Use of interfaces and @Qualifier/profiles make customization easy. |
| **Difficult to Learn and Use** | Steep learning curve due to multiple complex concepts like home/remote interfaces. | Simple programming model using POJOs and annotations. | Developers can focus on business logic. |
| **Resource-Intensive** | High memory and CPU footprint due to full app server dependency. | Lightweight containers reduce resource usage significantly. | Faster startup and fewer server resources needed. |
| **Limited Integration** | Integration with legacy or 3rd party systems was difficult. | Provides easy integration with JDBC, JPA, JMS, Kafka, REST, SOAP, etc. | Uses templates, adapters, and abstraction layers. |
| **Vendor Lock-in** | Dependent on specific app servers (JBoss, WebLogic, etc.). | Spring is open-source and works with any container or framework. | No vendor lock-in, portable across environments. |

## 🔹 **Spring Ecosystem**

Spring is not just a framework but a complete ecosystem. The Spring ecosystem is a collection of open-source tools and frameworks designed to simplify and accelerate Java application development, particularly for enterprise-level applications. It's built upon the foundation of the Spring Framework. It includes:

* **Core Container:** The central part of Spring, responsible for IoC (Inversion of Control) and DI (Dependency Injection), which manages the life cycle and dependencies of objects.
* **ApplicationContext**: The core container interface that provides additional enterprise services such as event propagation, internationalization, and resource loading, extending BeanFactory.
* **Beans Module**: Manages Spring Beans, their configurations, lifecycle, and dependencies via Java Config or XML.
* **AOP (Aspect-Oriented Programming):** A module that provides cross-cutting concerns such as logging, transaction management, and security in a modular fashion, separating them from business logic.
* **Spring Boot**: A Spring module that simplifies setup of Spring applications by embedding a web server and providing production-ready features like auto-configuration.
* **Spring Data:** Provides easy access to data stores (relational and non-relational) through common abstractions and simplifies CRUD operations.
* **Spring Cloud**: A collection of tools for building microservices-based applications, including service discovery, distributed configuration, and load balancing
* **Web Layer (Spring MVC):** Provides a powerful framework for building web applications using the Model-View-Controller (MVC) pattern, along with RESTful web services.
* **ORM Module:** Integrates with ORM frameworks like Hibernate, JPA, MyBatis, etc., to simplify object-relational mapping and persistence management.



# **2. Spring Core Concepts**

## 🔹 **Inversion of Control**

*Inversion of Control (IoC) is a* ***design principle*** *where the control of object creation and dependency management is transferred from the application code to a container or framework.*

* Traditionally, you create objects manually (new keyword) and manage their dependencies.
* With IoC, the container (like Spring) creates, configures, and manages objects for you.

***Real-world Analogy:***

Imagine you're at a restaurant:

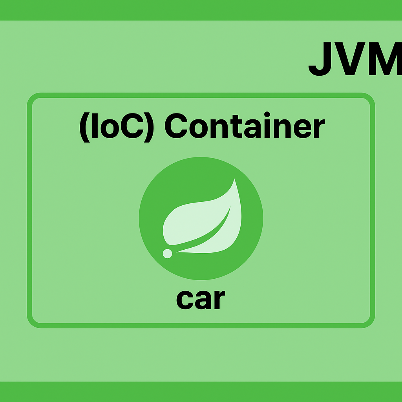
* Without IoC: You go into the kitchen and cook your food yourself.
* With IoC: You place an order, and the chef (Spring Container) prepares and serves it to you!

**How Spring Implements IoC**

Spring uses IoC Containers to manage the complete lifecycle of beans (objects):

* Creation: Spring instantiates your objects.
* Configuration: Spring sets the necessary properties and dependencies.
* Lifecycle Management: Spring controls the initialization and destruction.

In Spring, IoC is mainly implemented through Dependency Injection (DI), where dependencies are injected into classes rather than classes fetching them themselves.



## 🔹 **Dependency Injection**

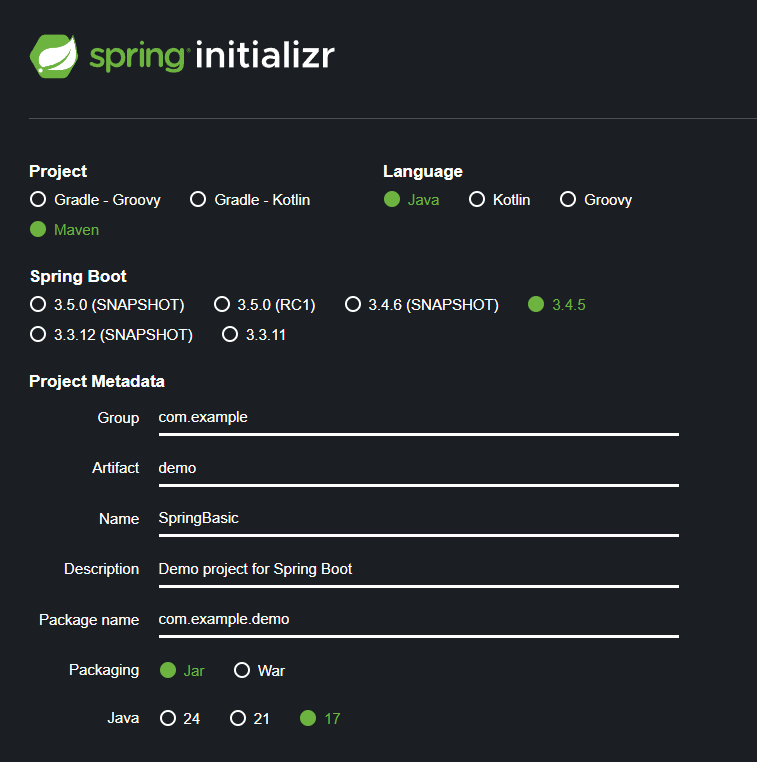
***Dependency Injection (DI)*** *is a design pattern where an object's dependencies are provided to it from an external source rather than the object creating them itself.*

IoC is a broader principle where control of object creation and flow is given to a container or framework, while Dependency Injection is a specific way to implement IoC by supplying dependencies from outside the class.

*CODE EXAMPLE:*

Implementing DI in spring project.

1. Create a basic springBoot project via <https://start.spring.io/>. Import the project to IDE



2. Create Car class, whose object will be created via DI

@Component //makes class a bean, this will be handled by IOC container  
public class Car{  
 public void trip(){  
 System.*out*.println("Lets go for long drive....");  
 }  
}

3. Create context object and use it to initiate bean (here car object).

@SpringBootApplication  
public class SpringBasicApplication {  
  
 public static void main(String[] args) {  
 ApplicationContext context = SpringApplication.*run*(SpringBasicApplication.class, args);  
 //Car alto = new Car(); manual creation  
 Car alto = context.getBean(Car.class); //Dependency Injection  
 alto.trip();  
 }  
}

## 🔹 **How IoC and DI really helps??**

On the surface, creating objects using new seems simple enough, but as your application grows in size and complexity, it leads to tightly coupled code, which is hard to maintain, test, and scale.

Eg: **Traditional way**

class Engine {  
 void start() {  
 System.*out*.println("Engine Started");  
 }  
}  
  
class Car {  
 private Engine engine;  
  
 Car() {  
 this.engine = new Engine(); // tightly coupled  
 }  
  
 void drive() {  
 engine.start();  
 System.*out*.println("Car is driving...");  
 }  
}

* Car creates its own Engine.
* What if you want to use a TurboEngine instead of a regular one? You have to modify the Car class.
* What if you want to unit test Car with a mock engine? It's tough.

**Spring way**

@Component  
class Engine {  
 void start() {  
 System.*out*.println("Engine Started");  
 }  
}  
  
@Component  
class Car {  
 private final Engine engine;  
  
 @Autowired  
 public Car(Engine engine) {  
 this.engine = engine;  
 }  
  
 void drive() {  
 engine.start();  
 System.*out*.println("Car is driving...");  
 }  
}

* Car doesn’t care how the Engine is created — it just uses it.
* You can inject a different implementation (TurboEngine, MockEngine) without changing the Car class.
* Spring manages the lifecycle and injection.

Yes, This can even be achieved using Interfaces, but then how DI is better than Interface + new keyword

**Lifecycle & Scope**

* Interface + new: you decide if it’s new per call, cached in a static field, pooled, etc.
* Spring: easily switch between singleton, prototype, request-scoped, session-scoped beans via simple configuration.

**Configuration externalized**

* Swap implementations (e.g. StripePaymentService ↔ PaypalPaymentService) by changing an annotation or properties file—no code change.
* Interfaces alone still require code or factory changes to pick one implementation over another.

**Cross-cutting concerns (AOP)**

* Transactions, caching, security, logging – all applied transparently around any bean method via aspects.
* With plain interfaces you’d have to manually wrap or decorate every call.

**Environment-specific wiring**

* Dev vs. QA vs. Prod profiles: load entirely different beans for different environments without touching application code.
* Interfaces + factories would require conditional logic sprinkled throughout your codebase.

Thus with DI we have:

1. **Loose Coupling:** Changes in one class do not affect others.
2. **Easier Testing:** You can inject mocks without hacking around.
3. **Better Configuration:** All dependencies are managed from one place.
4. **More Scalable:** You can swap out real implementations easily.

# **3. Basic Spring terms**

## 🔹 **Bean**

***Bean*** *is just* ***a Java object*** *that is managed by the Spring container.*

* "Managed" means: Spring is responsible for creating, initializing, configuring, injecting dependencies into, and destroying the object at the right time.
* Beans live inside the ApplicationContext (the big container holding all beans).

**To create bean in spring:** Use @Component and friends (@Service, @Repository, @Controller)

//Spring automatically scans for classes with these annotations  
@Component  
public class PaymentService {  
 // some code  
}

@Service  
public class OrderService {  
 // some code  
}

* **@Service** ➔ Marks a class as a business logic layer bean (performs core operations, like processing orders, payments, etc.).
* **@Repository** ➔ Marks a class as a data access layer bean (talks to the database, handles CRUD operations, translates exceptions).
* **@Controller** ➔ Marks a class as a web layer bean (handles HTTP requests, returns web pages or data to the user).

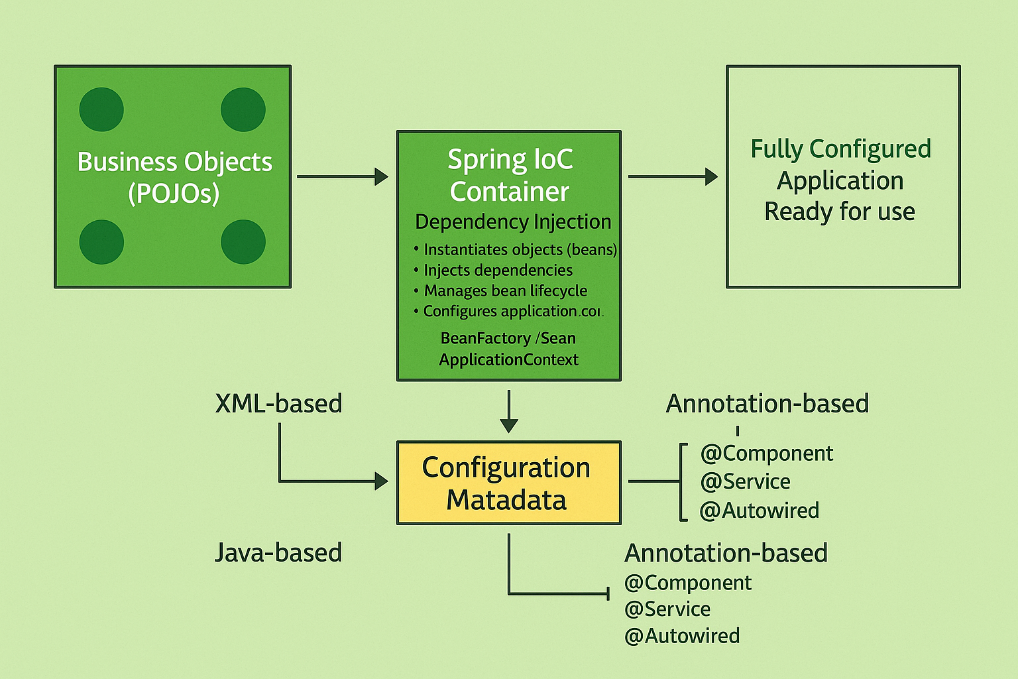
All three are specialized versions of @Component, meaning Spring will automatically detect and register them as Beans.

## 🔹 **IoC container**

*The* ***IoC (Inversion of Control)******Container*** *in Spring is a framework component that manages the creation, configuration, and lifecycle of application objects (beans) by injecting their dependencies automatically.*

IoC (Inversion of Control) Container in Spring is the brain that:

* Creates objects (beans),
* Manages their full life cycle,
* Injects dependencies,
* Configures them based on your instructions (annotations, XML, Java Config),
* And destroys them when no longer needed.



**How IoC containers works**

1. Start application → Spring Boot runs or you load Spring manually.
2. Scan classes → Finds @Component, @Service, @Repository, etc.
3. Instantiate beans → Creates objects from these classes.
4. Inject dependencies → Fulfills dependencies inside these beans.
5. Manage lifecycle → Initializes, monitors, and destroys beans properly.

**Types of IoC container**

BeanFactory vs ApplicationContext

| **Feature** | **BeanFactory** | **ApplicationContext** |
| --- | --- | --- |
| **Definition** | Basic container that provides fundamental DI. | Advanced container with additional features. |
| **Eager/Lazy Loading** | Lazy initialization (creates beans only when requested). | Eager initialization (creates all singleton beans at startup). |
| **Internationalization** | Not supported. | Supported (message source). |
| **Event Propagation** | No support. | Built-in support for publishing and listening to events. |
| **AOP Integration** | Minimal. | Full AOP capabilities. |
| **Common Usage** | Used for lightweight applications or memory-critical apps. | Standard choice for almost all Spring applications. |

Spring boot by defaults uses applicationContext. Also, ApplicationContext extends BeanFactory. ApplicationContext inherits everything that BeanFactory can do (basic bean creation, dependency injection).

PLUS, ApplicationContext adds more advanced features like:

* Event publishing (ApplicationEventPublisher)
* Message resource handling (for i18n — internationalization)
* Automatic BeanPostProcessors (custom initialization logic)
* Application lifecycle management (shutdown hooks, etc.)
* AOP (Aspect-Oriented Programming) support

## 🔹 **Autowiring**

***Autowiring*** *means automatically injecting the required dependencies into a class without manually writing code to do so.*

Instead of developer saying:

Car car = new Car(new Engine());

Spring will find an Engine for your Car and will inject it automatically. It is achieved using @Autowired annotation.

*CODE EXAMPLE:*

1. Create a class which you want to use in some other class, and make it a component

@Component  
public class Engine {  
 public void start(){  
 System.*out*.println("Engine Started...");  
 }  
}

2. Engine is required in Taxi class, so autowire engine with Taxi

@Component  
public class Taxi {  
 @Autowired  
 private Engine engine;  
  
 public void drive(){  
 engine.start();  
 System.*out*.println("Let's drive...");  
 }  
}

**How Autowiring works internally**

1. Spring scans your project for @Component, @Service, @Repository, etc.
2. Spring creates instances (beans) of these classes.
3. When Spring finds @Autowired:

* It looks for a matching bean by type (first).
* If multiple beans of the same type, it looks by name.
* If confused, Spring throws an error (unless you handle it with @Qualifier or make it @Primary).

1. Spring injects the matching bean into the required place.

## 🔹 **Types of Dependency Injection**

Spring supports 3 types of Dependency Injection:

**1. Field Injection**

@Component  
public class Student {  
 @Autowired  
 private Course course;  
}

Easy, but not recommended as:

* Violates good design principles (no clear dependencies via constructors).
* Difficult to test (need reflection/mocks).
* Makes classes tightly coupled to the framework.

**2. Constructor Injection**

@Component  
public class Student {  
 private final Course course;  
  
 @Autowired //use of this annotation is not compulsory for constructor injection

public Student(Course course) {  
 this.course = course;  
 }  
}

Advantage:

* Promotes immutability (final fields).
* Ensures mandatory dependencies are provided.
* Easy for unit testing (can pass mocks directly).

**3. Setter Injection**

@Component  
public class Student {  
 private Course course;  
  
 @Autowired  
 public void setCourse(Course course) {  
 this.course = course;  
 }  
}

Advantage: Useful for optional dependencies. Flexible — can inject after object creation.

Disadvantage: Object can exist in a partially initialized state, if dependencies are not injected. Also it is harder to enforce mandatory fields.

# **4. XML-Based Configuration in Spring**

Before Spring annotations and Java config styles became popular, XML was the primary way to configure Spring applications. It’s still important because many legacy systems and interviews expect you to know this.

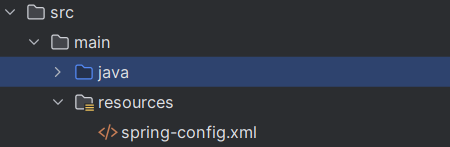
* Separation of configuration from code.
* No need to modify source code for wiring beans.
* Flexible and dynamic — can change wiring by just updating XML.
* Useful when annotations are restricted (older systems).

*CODE EXAMPLE:*

1. Create a class for which you want to create a bean

package org.example;  
public class Book {  
 public Book(){  
 System.*out*.println("Book Created");  
 }  
 public void read(){  
 System.*out*.println("Started reading the book");  
 }  
}

2. Create a config xml file in src/main/resources, also need to add default content. Also need to add <bean> tag with id and path of class.



<?xml version="1.0" encoding="UTF-8"?>  
<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  
 https://www.springframework.org/schema/beans/spring-beans.xsd">  
 <!--This was default content to be added-->  
  
 <!-- Define Book Beans -->  
 <!-- Property or constructor injection -->  
 <bean id="book1" class="org.example.Book">  
 </bean>  
  
</beans>

3. Update the main code, change applicationContext to ClassPathXmlApplicationContext and add config file name here. Now you can get book object by providing id name of bean

package org.example;  
import org.springframework.context.\*;  
import org.springframework.context.support.ClassPathXmlApplicationContext;  
  
public class App   
{  
 public static void main( String[] args )  
 {  
 ApplicationContext context = new ClassPathXmlApplicationContext("spring-config.xml");  
 Book book = (Book) context.getBean("book1");   
 book.read();  
 }  
}

output:

Book Created  
Started reading the book

**Note:** Even when book is not referenced, when we create context, all it’s bean(without prototype context) gets created. As ApplicationContext do early loading. Eg:

public class App   
{  
 public static void main( String[] args )  
 {  
 ApplicationContext context = new ClassPathXmlApplicationContext("spring-config.xml");  
 }  
}

output:

Book Created

To do lazy initialization of bean, can do

<bean id="book1" class="org.example.Book" lazy-init="true">  
</bean>

## 🔹 **Scope of Bean**

| **Scope** | **Description** |
| --- | --- |
| **singleton** | Only one instance per Spring container (default). |
| **prototype** | New instance created each time the bean is requested. |
| **request** | New instance for every HTTP request (Web applications only). |
| **session** | New instance for every HTTP session. |
| **application** | Single instance per web application context. |

request, session and application are meaningful only in Spring MVC apps.

**Singleton Scope**

Default scope of bean. Object is created when applicationContext loads. Only single object if created.

Eg:

public class App   
{  
 public static void main( String[] args )  
 {  
 ApplicationContext context = new ClassPathXmlApplicationContext("spring-config.xml");  
 Book book = (Book) context.getBean("book1");  
 book.author="abc";  
 System.*out*.println(book.author);  
 //2nd object creation  
 Book book2 = (Book) context.getBean("book1"); //references book only  
 System.*out*.println(book2.author);  
 }  
}

Output:

Book Created  
abc   
abc

**Prototype Scope**

Object is created only when it is called via getBean. Creates new object whenever we do context.getBean.

<bean id="book1" class="org.example.Book" scope="prototype">  
</bean>

With same above code, we will get output as:

Book Created  
abc  
Book Created  
null

## 🔹 **Setter Injection**

Setter injection means injecting values by calling setter methods on a bean.Here, actually setter is called to set values of given fields.

<bean id="book1" class="org.example.Book">  
 <property name="author" value="abc"/>  
 <property name="title" value="Best book ever"/>  
</bean>

* name = name of the setter (without set).
* value = the actual value to inject.

Above changes are for below pojo

package org.example;  
public class Book {  
 private String title;  
  
 public String getAuthor() {  
 return author;  
 }  
  
 public void setAuthor(String author) {  
 this.author = author;  
 }  
  
 private String author;  
   
 public String getTitle() {  
 return title;  
 }  
  
 public void setTitle(String title) {  
 this.title = title;  
 }  
}

This name, value works only with primitive types. For cases where object is injected as bean. Create bean for ubject being used, and ref to access it.

*CODE EXAMPLE:*

1. Create a class BookPublisher, which will be called in Book bean

public class BookPublisher {  
 public BookPublisher() {  
 System.*out*.println("Publisher called");  
 }  
}

2. Create getters and setters as we will do setter injection.

package org.example;  
public class Book {  
 private BookPublisher pub;  
  
 public BookPublisher getPub() {  
 return pub;  
 }  
  
 public void setPub(BookPublisher pub) {  
 this.pub = pub;  
 }  
}

3. Declare everything in xml file. Create a bean for BookPublisher. And use name = name of variable given in book and ref= id value given in bean declaration

<!--This was default content to be added-->  
<bean id="book1" class="org.example.Book">  
 <property name="pub" ref="publisher"/>  
</bean>  
  
<bean id="publisher" class="org.example.BookPublisher">  
</bean>

**Injecting Collection**

For injecting collections via xml in given file:

public class ShoppingCart {  
 private List<String> items;  
  
 public void setItems(List<String> items) {  
 this.items = items;  
 }  
}

Do

<bean id="shoppingCart" class="com.example.ShoppingCart">  
<property name="items">  
 <list>  
 <value>Apples</value>  
 <value>Bananas</value>  
 <value>Oranges</value>  
 </list>  
</property>  
</bean>

For map:

<bean id="employeeDirectory" class="com.example.EmployeeDirectory">  
<property name="employees">  
 <map>  
 <entry key="101" value="John Doe"/>  
 <entry key="102" value="Jane Smith"/>  
 </map>  
</property>  
</bean>

For book example above, if we need list of objects(eg books) for our bookstore object.

<bean id="bookstore" class="com.example.Bookstore">  
<property name="books">  
 <list>  
 <ref bean="book1"/>  
 <ref bean="book2"/>  
 </list>  
</property>  
</bean>

## 🔹 **Constructor Injection**

Sometimes you want object fully built at construction time, so you use the constructor. Eg:

public class Book {  
 private String title;  
 private int price;  
 public Book(String title, int price) {  
 this.title = title;  
 this.price = price;  
 }  
}

To inject values via xml

<bean id="book1" class="org.example.Book">  
 <constructor-arg value="abc"/>  
 <constructor-arg value="130"/>  
</bean>

Order matters if no name or index attribute is used.

Optional attributes:

* index = specify parameter position.
* name = specify parameter name (Java 8+ and compiled with -parameters).

<bean id="book1" class="org.example.Book">  
 <constructor-arg index="1" value="130"/>  
 <constructor-arg index="0" value="abc"/>  
</bean>

**Nested Bean**

If a bean depends on another bean, you can define a bean inside another <bean> using <property> or <constructor-arg>.

public class Computer {  
 private Processor processor;  
 public void setProcessor(Processor processor) {  
 this.processor = processor;  
 }  
}  
  
public class Processor {  
 private String type;  
 public void setType(String type) {  
 this.type = type;  
 }  
}

xml:

<bean id="computer" class="com.example.Computer">  
<property name="processor">  
 <bean class="com.example.Processor">  
 <property name="type" value="Intel i7"/>  
 </bean>  
</property>  
</bean>

## 🔹 **Working with Interfaces**

Lets say, we have a interface

public interface PaymentService {  
 void pay();  
}

and its 2 implementations

public class CreditCardPaymentService implements PaymentService {  
 @Override  
 public void pay() {  
 System.*out*.println("Paid using Credit Card");  
 }  
}

public class PaypalPaymentService implements PaymentService {  
 @Override  
 public void pay() {  
 System.*out*.println("Paid using PayPal");  
 }  
}

For client code, where it needs PaymentService object

public class PaymentClient {  
 private PaymentService paymentService;  
  
 // Setter injection  
 public void setPaymentService(PaymentService paymentService) {  
 this.paymentService = paymentService;  
 }  
  
 public void doPayment() {  
 paymentService.pay();  
 }  
}

For XML:

<!-- Beans for Implementations -->  
<bean id="creditCardPayment" class="com.example.CreditCardPaymentService" />  
  
<bean id="paypalPayment" class="com.example.PaypalPaymentService" />  
  
<!-- Client Bean -->  
<bean id="paymentClient" class="com.example.PaymentClient">  
 <property name="paymentService" ref="creditCardPayment"/>  
</bean>

In these case, whenever we want to change paymentService to payPal, we can just do by changing ref value in xml.

## 🔹 **Autowiring**

*Autowiring in Spring automatically injects the required dependencies into a bean without explicitly specifying them in the configuration.*

**Autowiring Type**

* byName: Spring injects dependency based on matching bean id and the property name.
* byType: Spring injects dependency based on matching bean type (class/interface).

Eg: in above example of paymentService, in PaymentClient PaymentService attribute name is paymentService. So we can give same name as id for PaypalPayment, so that will automatically get injected

<!-- Client Bean -->  
<bean id="paymentClient" class="com.example.PaymentClient" autowire="byName">  
</bean>  
<!-- Beans for Implementations -->  
<bean id="creditCardPayment" class="com.example.CreditCardPaymentService" />  
  
<bean id="paymentService" class="com.example.PaypalPaymentService" />

Eg2: If we are using autowire byType, we have 2 beans of type PaymentService, so in this case we can make one bean primary and that will be

<!-- Client Bean -->  
<bean id="paymentClient" class="com.example.PaymentClient" autowire="byType">  
</bean>  
<!-- Beans for Implementations -->  
<bean id="creditCardPayment" class="com.example.CreditCardPaymentService" />  
  
<bean id="paypalPayment" class="com.example.PaypalPaymentService" primary="true"/>