Class-19 : Understanding Spring Core

by Pial Kanti Samadder

Why Learn Spring Boot?

Simplifying Modern Java Development

Spring Is Everywhere

Spring powers millions of applications worldwide—from streaming and e-commerce to finance. Trusted by top tech giants like Google, Amazon, and Microsoft.

Spring Is Fast

Optimized startup, reactive support, and built-in tools like LiveReload make Spring one of the fastest frameworks for modern apps.

Spring Is Secure

Built-in Spring Security and active vulnerability management ensure robust protection for your apps and user data.

Spring Is Flexible

With IoC, DI, and rich integrations, Spring lets you build anything—from simple APIs to large, cloud-native microservices.

Spring Is Productive

Spring Boot removes boilerplate and simplifies configuration, letting developers focus on business logic and deliver faster.

Spring Has a Great Community

A global, vibrant community provides tutorials, guides, and open-source support to help you learn, grow, and build with confidence.

Spring vs Spring Boot

Spring Framework

- A comprehensive **framework** providing features like **IoC**, **DI**, **AOP**, **Transaction Management**, and **MVC**.
- Requires **manual configuration** for components, dependencies, and web servers.
- Flexible but **setup-heavy** — you decide and wire everything.
- Best suited for **custom, large enterprise projects** needing deep control.

Spring Boot

- An **opinionated extension** of the Spring Framework that **simplifies setup and development**.
- Uses **auto-configuration** and **starter dependencies** to reduce boilerplate.
- Comes with an **embedded web server (Tomcat/Jetty)** — no need to deploy WAR files.
- Ideal for **rapid development**, **microservices**, and **cloud-ready** applications.

In Simple Terms

	Spring Framework	Spring Boot
Configuration	Manual (XML/Java Config)	Auto-configured
Dependency Setup	Custom Maven/Gradle setup	Starter dependencies
Web Server	External (Tomcat, etc.)	Embedded
Complexity	High for beginners	Easy to start
Use Case	Enterprise-scale, custom solutions	Fast development, microservices

Challenges in Traditional Java Development

- **Manual Dependency Management**
  Developers had to manually create and wire objects, increasing coupling and reducing flexibility.
- **Complex Configuration**
  XML files scattered across projects made setup error-prone and difficult to maintain.
- **Tedious Deployment**
  Setting up and managing web servers manually slowed down delivery and increased friction.
- **Hard to Maintain & Test**
  Large applications became hard, with poor testability and long feedback cycles.

From Manual Wiring to Inversion of Control (IoC)

Traditional Java (Manual Wiring)

- Objects are created manually, e.g., `OrderService orderService = new OrderService();`
- Classes are tightly coupled, making testing and maintenance difficult.
- Developers manage dependencies themselves, leading to “configuration hell.”

Spring’s Approach (IoC & Managed Beans)

- Spring **manages object creation** and wiring through its **IoC container**.
- Developers define *what* they need; Spring **provides it automatically** via Dependency Injection (DI).
- Improves **modularity, testability, and maintainability**.

What is Inversion of Control (IoC)?

Inversion of Control (IoC) is a design principle in which the **control of object creation and management is transferred from the developer to the framework**.

In simple terms:

> **Inversion of Control is when a framework or container, instead of your code, is in charge of the program flow.**

Normally, your code decides **when** and **how** methods are called. With IoC, you **hand over control** to the framework, and it calls your code when appropriate.

Traditional Java Example (No IoC)

```
public class PaymentService {
    private OrderService orderService;

    public PaymentService() {
        this.orderService = new OrderService(); // YOU create the dependency
    }

    public void processPayment() {
        orderService.createOrder();
        System.out.println("Payment processed");
    }
}
```

What's happening here:

- `PaymentService` **decides when to create** `OrderService`.
- You are in **full control**.
- Problem: If you want to test `PaymentService`, you **cannot easily replace** `OrderService` with a mock.

IoC Version (Spring manages objects)

```
@Component
public class OrderService {
    public void createOrder() {
        System.out.println("Order created");
    }
}

@Component
public class PaymentService {
    private final OrderService orderService;

    @Autowired
    public PaymentService(OrderService orderService) {
        this.orderService = orderService; // Spring injects it
    }

    public void processPayment() {
        orderService.createOrder();
        System.out.println("Payment processed");
    }
}
```

What's happening here:

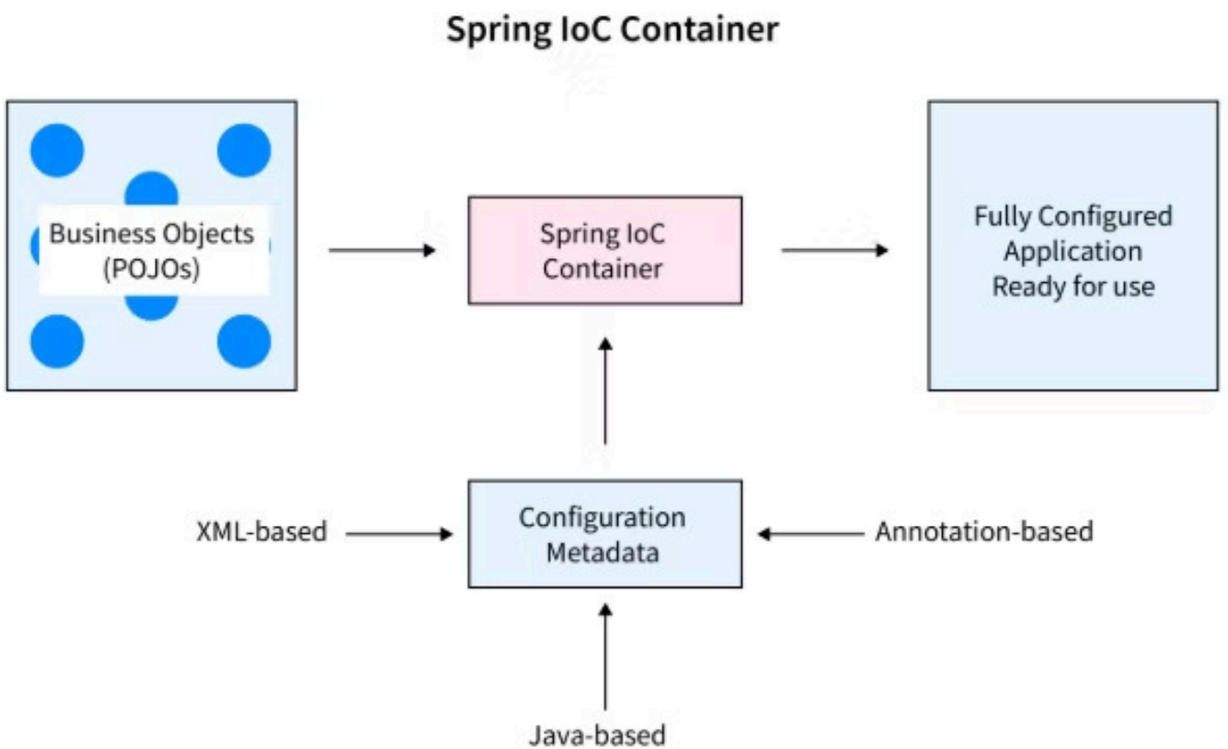
- You **do not create** `OrderService` **manually**.
- Spring **instantiates it** and injects it into `PaymentService`.
- Control is **inverted**: Spring decides **when and how to create objects**.
- Benefit: Easily replace `OrderService` with a mock for testing.

The control is inverted - it calls me rather me calling the framework.

This phenomenon is *Inversion of Control* (**also known as the Hollywood Principle - *"Don't call us, we'll call you"***).

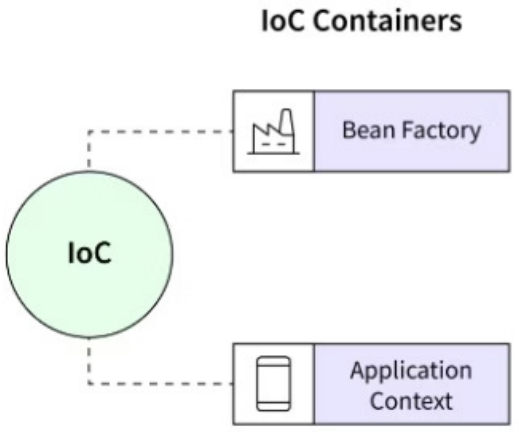
Spring IoC Container

- **Core component** that manages **Spring beans** (objects) in your application.
- Responsibilities:
  a. **Bean Creation:** Instantiates objects defined in the context.
  b. **Dependency Injection:** Provides required dependencies to beans.
  c. **Bean Lifecycle Management:** Handles initialization, destruction, and callbacks.



Types of IoC Containers

There are two main types of IoC (Inversion of Control) containers:



1. **BeanFactory:**
   - The BeanFactory is the simplest form of the Spring IoC container. It is the foundation interface for the Spring IoC container and provides the fundamental features of the container, including the ability to manage beans and their dependencies.
   - **Key Characteristics:**
     - **Lazy loading of beans:** Beans are only created when requested.
     - **Lightweight:** Suitable for resource-constrained environments.
     - **Basic container features:** Provides essential IoC functionality.
2. **ApplicationContext:**
   - The ApplicationContext is a more advanced and feature-rich container than BeanFactory. It extends the BeanFactory interface and adds additional functionalities such as event propagation, AOP (Aspect-Oriented Programming), and integration with other Spring modules.
   - **Key Characteristics:**
     - **Eager loading of beans:** Beans are preloaded on container startup.
     - **Full container features:** Provides all BeanFactory features and more features.
     - **Suitable for enterprise applications:** Offers a wide range of functionalities for complex applications.
     - **Supports multiple configurations:** XML-based, annotation-based, and Java-based configurations.

Container Configuration

The configuration of the Spring IoC container can be done using XML-based configuration files, Java-based configuration classes, or a combination of both. These configurations define the beans and their relationships (dependencies) within the application.

Based on the configuration style, IoC containers can be categorized into three types:

1. **XML-based Configuration:**
   - Configuration uses XML files where beans and their dependencies are defined.
   - Example: Spring XML configuration files.
2. **Annotation-based Configuration:**
   - Configuration is done using annotations (e.g., `@Component`, `@Autowired`) in the source code.
   - Example: Spring's component scanning and annotations.
3. **Java-based Configuration:**
   - Configuration is done using Java classes (often annotated with `@Configuration`) that define beans and their relationships.
   - Example: Spring's JavaConfig.

Note:

`@Configuration` Indicates that a class declares **one or more** `@Bean` **methods and may be processed by the Spring container to generate bean definitions and service requests for those beans at runtime.**

`@Component` Indicates that an annotated class is a "component". **Such classes are considered as candidates for auto-detection when using annotation-based configuration and classpath scanning**.

Bean Overview

Here's a definition of beans in the ***Spring Framework documentation***:

> *In Spring, the objects that form the backbone of your application and that are managed by the Spring IoC container are called beans. A bean is an object that is instantiated, assembled, and otherwise managed by a Spring IoC container.*

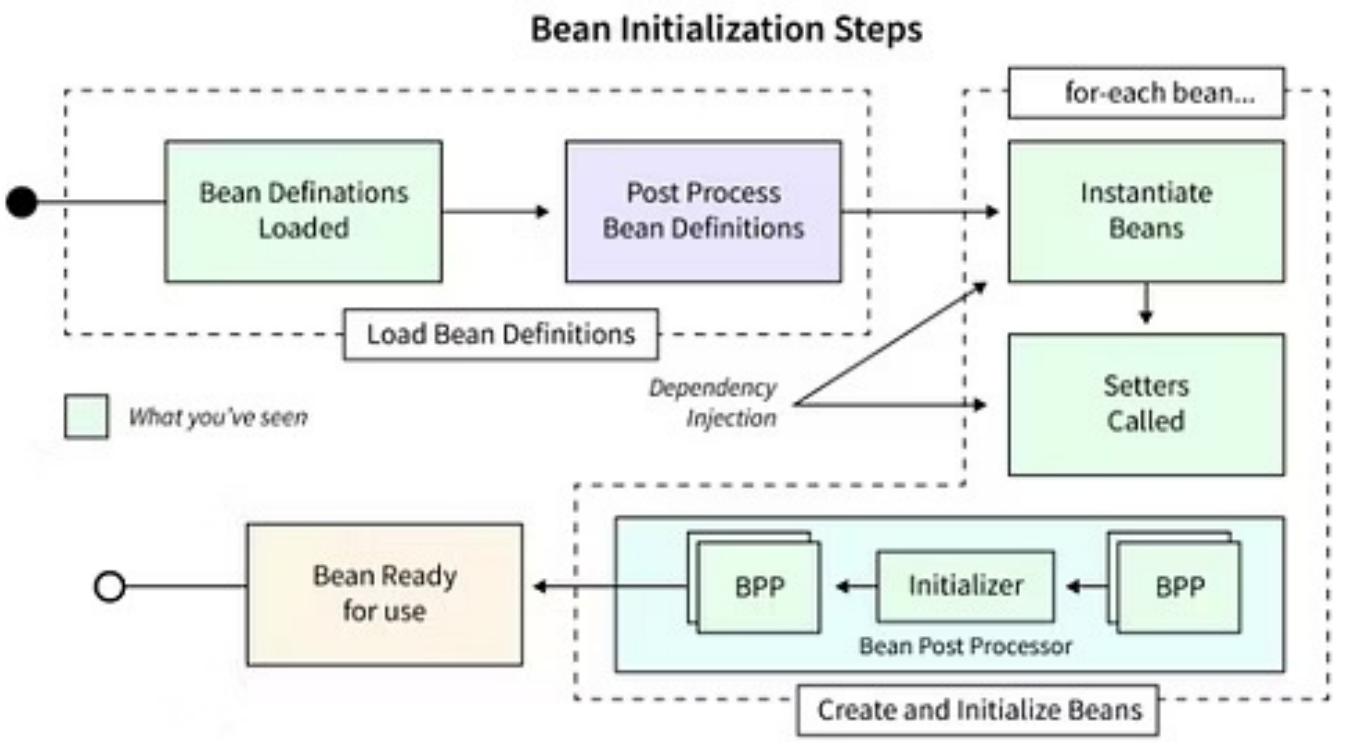
Beans are defined in the Spring configuration and are typically Java classes annotated with `@Component` or configured in XML or Java-based configuration.

Bean Scopes

Beans can be defined to be deployed in one of a number of scopes. **The Spring Framework supports six scopes**, four of which are available only if you use a web-aware `ApplicationContext`.

Scope	Description
singleton	(Default) Scopes a single bean definition to a single object instance for each Spring IoC container.
prototype	Scopes a single bean definition to any number of object instances.
request	Scopes a single bean definition to the lifecycle of a single HTTP request. That is, each HTTP request has its own instance of a bean created off the back of a single bean definition. Only valid in the context of a web-aware Spring `ApplicationContext`.
session	Scopes a single bean definition to the lifecycle of an HTTP `Session`. Only valid in the context of a web-aware Spring `ApplicationContext`.
application	Scopes a single bean definition to the lifecycle of a `ServletContext`. Only valid in the context of a web-aware Spring `ApplicationContext`.
websocket	Scopes a single bean definition to the lifecycle of a `WebSocket`. Only valid in the context of a web-aware Spring `ApplicationContext`.

Spring Bean Life Cycle Stages



1. **Bean Definition Acquisition**: The bean is defined using Java config or XML bean configurations.
2. **Bean Creation and Instantiation**: As soon as the bean is defined, it will be instantiated, initialized with properties, and then fully managed within the ApplicationContext.
3. **Populating Bean Properties**: The Spring IoC container populates the bean properties including **id**, **scope**, and **property values** based on the bean definition provided in your Java config class or XML.
4. **Post-Initialization**: Spring provides **Aware** interfaces to access bean definition metadata details and callback methods to hook into the bean life cycle, allowing for custom logic before the bean's initialization methods.
5. **Ready to Serve**: At this stage, all required dependencies are injected, and all the Aware implemented callback methods are executed. The bean is ready to serve.
6. **Pre-Destroy**: Before a bean is destroyed, Spring's callback methods allow for the execution of application-specific cleanup logic in the ApplicationContext.
7. **Bean Destroyed**: Now, as all the pre-destroy callback methods are executed, ApplicationContext will destroy/delete the bean from the JVM.