What is a Spring Bean?

In the **Spring Framework**, a **Spring Bean** is an object that is managed by the Spring **IoC (Inversion of Control) container**. These are the core components of a Spring application and are instantiated, configured, and wired together by the container. Essentially, a Spring Bean is a Java object that is created, maintained, and destroyed by the Spring container.

Key Characteristics of Spring Beans

1. **Managed by the Spring Container**:
    - Spring is responsible for creating and managing the lifecycle of beans, from instantiation to destruction.
2. **Configured via Annotations, XML, or Java Configuration**:
    - Beans can be defined using various configuration styles:
        - **Annotations** (**e.g., @Component, @Service, @Repository**)
        - **XML Configuration (<bean> tag in XML files)**
        - **Java-based Configuration (@Bean methods in @Configuration classes)**
3. **Dependency Injection**:
    - Spring uses **Dependency Injection (DI)** to provide bean dependencies to other beans, allowing for loose coupling and easier testing.
4. **Singleton by Default**:
    - By default, Spring beans are **singleton** scoped, meaning a single instance is created and shared across the application. However, Spring also supports other scopes like **prototype**, **request**, and **session**.

How to Define a Spring Bean

Spring provides three main ways to define and configure beans:

1. Using Annotations (@Component, @Service, etc.)

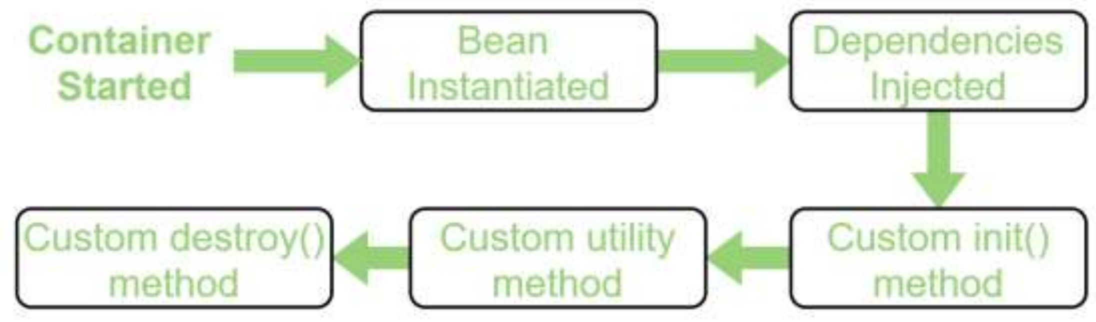
Annotations are the most modern and preferred way to declare beans in Spring applications.

Bean life cycle in Java Spring

-

The lifecycle of any object means when & how it is born, how it behaves throughout its life, and when & how it dies. Similarly, the bean life cycle refers to when & how the bean is instantiated, what action it performs until it lives, and when & how it is destroyed. In this article, we will discuss the life cycle of the bean.

Bean life cycle is managed by the spring container. When we run the program then, first of all, the spring container gets started. After that, the container creates the instance of a bean as per the request, and then dependencies are injected. And finally, the bean is destroyed when the spring container is closed. Therefore, if we want to execute some code on the bean instantiation and just after closing the spring container, then we can write that code inside the custom **init()** method and the **destroy()** method.
The following image shows the process flow of the bean life cycle.



Bean Life Cycle Process Flow

Note: We can choose a custom method name instead of **init()** and **destroy()**. Here, we will use init() method to execute all its code as the spring container starts up and the bean is instantiated, and destroy() method to execute all its code on closing the container.

Lifecycle of a Spring Bean

A Spring Bean goes through several stages during its lifecycle:

1. **Instantiation**: The bean is created by the container.
2. **Dependency Injection**: Properties are set and dependencies are injected.
3. **Initialization**: Custom initialization methods like **@PostConstruct** are called.
4. **Ready for Use**: The bean is ready for use in the application.
5. **Destruction**

Why Use Spring Beans?

1. **Dependency Management**: Spring beans promote loose coupling by allowing the container to inject dependencies, which makes it easier to manage dependencies between objects.
2. **Centralized Configuration**: All beans are defined and managed centrally, making it easier to configure and modify the application's behavior.
3. **Lifecycle Management**: Spring automatically handles the lifecycle of beans, including initialization and cleanup, ensuring efficient resource management.
4. **Aspect-Oriented Programming (AOP)**: Beans are often used in conjunction with AOP features in Spring, such as transaction management and logging.

Summary

- A **Spring Bean** is any Java object that is instantiated, managed, and destroyed by the Spring IoC container.
- Beans are typically defined using annotations **(@Component, @Service),** Java configuration **(@Bean**), or XML configuration.
- The **Spring container** manages the lifecycle of beans, handles dependency injection, and ensures efficient resource management.
- Understanding how to define and use beans is fundamental to developing Spring applications, enabling better organization, scalability, and testability of the code.

Spring beans are the building blocks of a Spring application, providing the core foundation for creating robust, maintainable, and scalable enterprise applications.

Spring Bean Lifecycle in the Context of the Example

The **Spring Bean Lifecycle** refers to the stages a bean goes through from creation to destruction, managed by the **Spring IoC container**. Here's how it applies to the Employee and Address beans in your example:

1. **Instantiation**:
    - The Spring IoC container reads the XML configuration (applicationContext.xml) and creates instances of the beans (Address and Employee).
2. **Property Injection**:
    - The container injects dependencies by calling setter methods:

        - It sets the id, name, and address properties of the Employee bean.
        - The address property is injected as a reference to the Address bean (a1 or address2).
3. **Post-Initialization (@PostConstruct or custom init method)**:
    - If there were any post-initialization methods (e.g., @PostConstruct), Spring would call them after setting properties. However, in this example, no specific initialization methods are defined.
4. **Bean Usage**:
    - The Employee bean is accessed via getBean in the Test class, and its display method is called to print details.
5. **Pre-Destruction (@PreDestroy or custom destroy method)**:
    - When the application context is closed (via context.close() or registerShutdownHook()), Spring would call any pre-destroy methods. In this case, no @PreDestroy or destroy methods are defined.
6. **Destruction**:
    - The Spring IoC container destroys the bean, releasing any resources if necessary.

Summary

The lifecycle stages in your example are **Instantiation**, **Dependency Injection**, **Bean Usage**, and **Destruction**. Custom initialization (@PostConstruct) and cleanup (@PreDestroy) are optional stages not included in this example but are part of the typical Spring bean lifecycle