**[Spring Bean Scopes](https://www.bing.com/ck/a?!&&p=d14312131ef12f6b361c609934e6fcfde04b78f4b7df6facf108ca3677d04cb6JmltdHM9MTc0OTQyNzIwMA&ptn=3&ver=2&hsh=4&fclid=120e6dd7-4668-660d-15c4-7938471767d7&psq=spring+bean+scopes&u=a1aHR0cHM6Ly9kb2NzLnNwcmluZy5pby9zcHJpbmctZnJhbWV3b3JrL3JlZmVyZW5jZS9jb3JlL2JlYW5zL2ZhY3Rvcnktc2NvcGVzLmh0bWw&ntb=1" \t "_blank)**

In the Spring framework, the scope of a bean defines its lifecycle and visibility within the application context. Spring supports six types of bean scopes: singleton, prototype, request, session, application, and websocket[1](https://www.bing.com/ck/a?!&&p=d14312131ef12f6b361c609934e6fcfde04b78f4b7df6facf108ca3677d04cb6JmltdHM9MTc0OTQyNzIwMA&ptn=3&ver=2&hsh=4&fclid=120e6dd7-4668-660d-15c4-7938471767d7&psq=spring+bean+scopes&u=a1aHR0cHM6Ly9kb2NzLnNwcmluZy5pby9zcHJpbmctZnJhbWV3b3JrL3JlZmVyZW5jZS9jb3JlL2JlYW5zL2ZhY3Rvcnktc2NvcGVzLmh0bWw&ntb=1)[2](https://www.bing.com/ck/a?!&&p=318e342ea874a2f5fa47ff3a91dfd5dbdd6abfdfc5554011bc52cfb267422f2eJmltdHM9MTc0OTQyNzIwMA&ptn=3&ver=2&hsh=4&fclid=120e6dd7-4668-660d-15c4-7938471767d7&psq=spring+bean+scopes&u=a1aHR0cHM6Ly93d3cuYmFlbGR1bmcuY29tL3NwcmluZy1iZWFuLXNjb3Blcw&ntb=1). The last four scopes are only available in a web-aware application context.

**Singleton Scope**

The singleton scope is the default scope in Spring. When a bean is defined with the singleton scope, the Spring container creates a single instance of that bean, and all requests for that bean name return the same object. Any modifications to the object will be reflected in all references to the bean. This scope is ideal for stateless beans.

Example:

@Bean

@Scope("singleton")

public Person personSingleton() {

return new Person();

}

**Prototype Scope**

A bean with the prototype scope will return a different instance every time it is requested from the container. This scope is suitable for stateful beans, where each instance maintains its own state.

Example:

@Bean

@Scope("prototype")

public Person personPrototype() {

return new Person();

}

**Request Scope**

The request scope creates a bean instance for a single HTTP request. Each HTTP request has its own instance of the bean, and the bean is discarded once the request is completed.

Example:

@Bean

@RequestScope

public HelloMessageGenerator requestScopedBean() {

return new HelloMessageGenerator();

}

**Session Scope**

The session scope creates a bean instance for an HTTP session. The same instance of the bean is returned for the entire session, and it is discarded when the session ends.

Example:

@Bean

@SessionScope

public HelloMessageGenerator sessionScopedBean() {

return new HelloMessageGenerator();

}

**Application Scope**

The application scope creates a bean instance for the lifecycle of a ServletContext. This scope is similar to the singleton scope but is shared across multiple servlet-based applications running in the same ServletContext.

Example:

@Bean

@ApplicationScope

public HelloMessageGenerator applicationScopedBean() {

return new HelloMessageGenerator();

}

**WebSocket Scope**

The websocket scope creates a bean instance for the lifecycle of a WebSocket session. The same instance of the bean is returned whenever it is accessed during the entire WebSocket session.

Example:

@Bean

@Scope(scopeName = "websocket", proxyMode = ScopedProxyMode.TARGET\_CLASS)

public HelloMessageGenerator websocketScopedBean() {

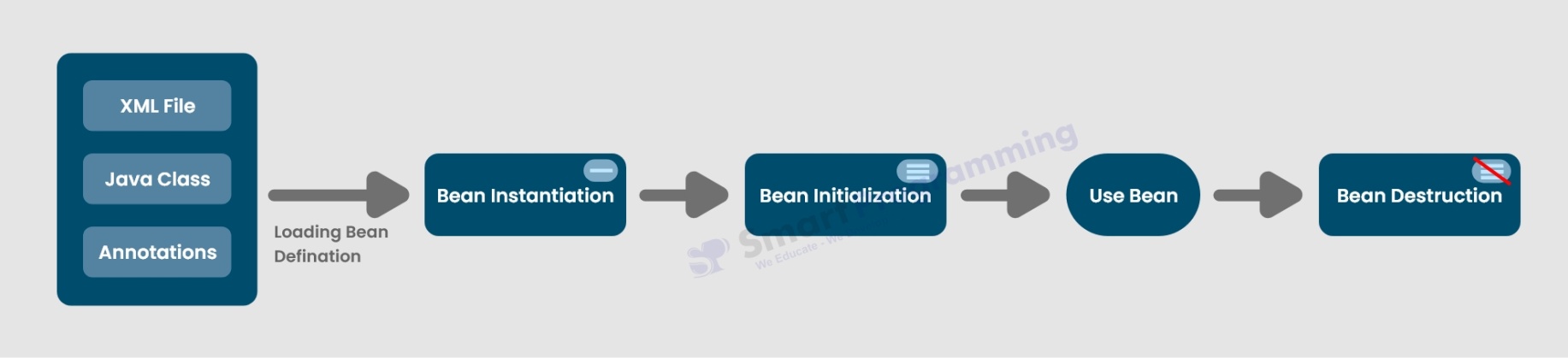
return new HelloMessageGenerator();

}

Understanding and using the appropriate bean scope is crucial for managing the lifecycle and visibility of beans in a Spring application. Each scope serves a specific purpose and is suited for different use cases[1](https://www.bing.com/ck/a?!&&p=79551cc71dd2579187c9ebc452966f5f32d64c5b3fbbbe50f9422615eeb88f9eJmltdHM9MTc0OTQyNzIwMA&ptn=3&ver=2&hsh=4&fclid=120e6dd7-4668-660d-15c4-7938471767d7&u=a1aHR0cHM6Ly9kb2NzLnNwcmluZy5pby9zcHJpbmctZnJhbWV3b3JrL3JlZmVyZW5jZS9jb3JlL2JlYW5zL2ZhY3Rvcnktc2NvcGVzLmh0bWw&ntb=1)[2](https://www.bing.com/ck/a?!&&p=6695acda69c5f5f800b9459152b62092f04e1036c2f33cca02f2464c08240004JmltdHM9MTc0OTQyNzIwMA&ptn=3&ver=2&hsh=4&fclid=120e6dd7-4668-660d-15c4-7938471767d7&u=a1aHR0cHM6Ly93d3cuYmFlbGR1bmcuY29tL3NwcmluZy1iZWFuLXNjb3Blcw&ntb=1).

### Spring Bean Life Cycle

#### Introduction

* In Spring, beans are the backbone of our application. They represent the objects that we have to create and manage within the Spring context.
* These beans go through a well-defined life cycle, where Spring handles everything from their creation to their destruction.
* Different stages of Bean Life Cycle are as below :-
  1. **Loading Bean Definitions :** Spring reads the configuration (XML, Java, or annotations) to understand how to create and configure beans, including their properties and dependencies.
  2. **Bean Instantiation :** Spring creates bean instances using constructors or factory methods. Then dependencies are injected through constructor OR setter OR property injection.
  3. **Bean Initialization :** After bean is instantiated and its properties are set, Spring performs initialization by calling an init() method OR afterPropertiesSet() method (from the InitializingBean interface) OR @PostConstruct annotation.
  4. **Bean Usage :** The fully initialized bean is now ready for use. It can interact with other beans or components in the application.
  5. **Bean Destruction :** Upon container shutdown, Spring destroys the bean. This can be done via custom destroy() method OR DisposableBean (interface) OR @PreDestroy annotation for cleanup.
* Below is the diagram representing Bean Life Cycle :-  
    
  

 Above Bean Life Stages are explained deeply as below :-

##### **1. Loading Bean Definitions :**

* **What Happens:** Spring first reads and understands the definitions of all the beans in the application. These definitions are like blueprints that tell Spring how to create and configure beans.
* **Where Definitions Come From:** These configurations can be provided by different ways as below:
  + **XML Configuration:** In the old days, beans were defined in XML files.
  + **Java Configuration:** With the rise of Java-based configuration, we can define beans using @Configuration and @Bean annotations.
  + **Annotations:** With annotations like @Component, @Service, etc., we can let Spring automatically discover and register beans.
* **What Spring Understands:** In this stage, Spring knows the class of each bean, its properties (like name, rollno, emailid in Student class) and its dependencies (other beans it needs).

##### **2. Bean Instantiation :**

* **What Happens:** Once Spring knows what beans it needs to create, it creates instances of those beans. This is the instantiation phase.
* **How Beans Are Created:** Beans can be created by different ways as below:
  + **Default Constructor:** A no-argument constructor is used.
  + **Static Factory Method:** A static method on a class that returns an instance of the bean.
  + **Instance Factory Method:** A non-static method on a separate factory class.
* **Dependency Injection:** During instantiation, dependencies are injected into the bean. This can be done in three ways:
  + **Constructor Injection:** Dependencies are provided through the constructor.
  + **Setter Injection:** Dependencies are set using setter methods.
  + **Property Injection:** Dependencies are set using XML tags or annotations like @Value (in Java-based config).
* **Example:** If we configure a Student bean with name, rollno, and emailid, Spring injects the property values (like name Deepak, rollno 102 and email id as deepak@gmail.com in this [example](https://smartprogramming.in/tutorials/spring-framework/spring-program-using-java-based-configurations.php)) in this phase (e.g., through setter methods or annotations).

##### **3. Bean Initialization :**

* **What Happens:** After a bean is created, it’s initialized by setting its properties to their configured values. This is where any external dependencies, such as values from a configuration file, are injected into the bean.
* **Different Ways to Initialize:** Beans are initialized by different ways as below:
  + **Using init() method:** We can define a custom initialization method in our bean.
  + **Using afterPropertiesSet():** This method from the InitializingBean interface is called to perform any necessary initialization.
  + **Using @PostConstruct:** A special annotation that indicates a method to be run after all properties are set.
* **Example:** Here the bean is prepared for use. This includes setting properties (if not already done in instantiation), performing validation, setting up database connections, initializing caches, configuring logging, and executing any custom initialization logic such as loading configuration files or establishing network connections.
* **Additional Functionality:**
  + **Aware Interfaces:** If the bean implements special interfaces, it can access the container or its environment. For example:
    - BeanNameAware: Allows the bean to know its own name.
    - ApplicationContextAware: Provides access to the Spring ApplicationContext.
    - BeanFactoryAware: Gives access to the BeanFactory (for bean creation).
* **Optional Customizations:** At this point, we can also use BeanPostProcessor to add custom logic before and after the bean’s initialization. This is useful for advanced scenarios like modifying beans dynamically or adding cross-cutting concerns (e.g., logging or security).

##### **4. Bean Usage :**

* **What Happens:** Now that the bean is initialized, it’s ready to be used. Other beans or components in the Spring application can access and use this bean. The bean is fully functional and serves its purpose in the application.

##### **5. Bean Destruction :**

* **What Happens:** When the application shuts down or the Spring container is closed, Spring cleans up by destroying the beans.
* **How Beans Are Destroyed:**
  + **Using destroy() method:** We can define a custom destruction method in the bean.
  + **Using DisposableBean interface:** This interface provides a destroy() method to clean up resources when the bean is no longer needed.
  + **Using @PreDestroy annotation:** This annotation marks a method to be called before the bean is destroyed.
* **Optional Aspects:** If the bean doesn't require any special clean-up, Spring will skip these methods.

[**Bean Life Cycle in Java Spring**](https://www.bing.com/ck/a?!&&p=6eb046658bb3c39af816bfb4bcf7650f24e398be4087eb6fb16cf01069509527JmltdHM9MTc0OTQyNzIwMA&ptn=3&ver=2&hsh=4&fclid=120e6dd7-4668-660d-15c4-7938471767d7&psq=spring+bean+life+cycle&u=a1aHR0cHM6Ly93d3cuZ2Vla3Nmb3JnZWVrcy5vcmcvYmVhbi1saWZlLWN5Y2xlLWluLWphdmEtc3ByaW5nLw&ntb=1)

[1](https://www.bing.com/ck/a?!&&p=6eb046658bb3c39af816bfb4bcf7650f24e398be4087eb6fb16cf01069509527JmltdHM9MTc0OTQyNzIwMA&ptn=3&ver=2&hsh=4&fclid=120e6dd7-4668-660d-15c4-7938471767d7&psq=spring+bean+life+cycle&u=a1aHR0cHM6Ly93d3cuZ2Vla3Nmb3JnZWVrcy5vcmcvYmVhbi1saWZlLWN5Y2xlLWluLWphdmEtc3ByaW5nLw&ntb=1)[2](https://www.bing.com/ck/a?!&&p=5acaa695efe150e6e011b225c58ddd1b7422b60720a4aff9c084652901858596JmltdHM9MTc0OTQyNzIwMA&ptn=3&ver=2&hsh=4&fclid=120e6dd7-4668-660d-15c4-7938471767d7&psq=spring+bean+life+cycle&u=a1aHR0cHM6Ly9ob3d0b2RvaW5qYXZhLmNvbS9zcHJpbmctY29yZS9zcHJpbmctYmVhbi1saWZlLWN5Y2xlLw&ntb=1)[3](https://www.bing.com/ck/a?!&&p=4a594ebdd45b1eb079cd45581c1402f8ab52f5a43b95d3c5848ad0a930e605d5JmltdHM9MTc0OTQyNzIwMA&ptn=3&ver=2&hsh=4&fclid=120e6dd7-4668-660d-15c4-7938471767d7&psq=spring+bean+life+cycle&u=a1aHR0cHM6Ly93d3cudHV0b3JpYWxzcG9pbnQuY29tL3NwcmluZy9zcHJpbmdfYmVhbl9saWZlX2N5Y2xlLmh0bQ&ntb=1)

The **bean life cycle** in Java Spring refers to the series of steps and methods that a bean goes through from its instantiation to its eventual destruction. This life cycle is managed by the Spring container, which is responsible for creating, configuring, and destroying beans according to the defined bean scope (singleton, prototype, etc.)[1](https://www.bing.com/ck/a?!&&p=6eb046658bb3c39af816bfb4bcf7650f24e398be4087eb6fb16cf01069509527JmltdHM9MTc0OTQyNzIwMA&ptn=3&ver=2&hsh=4&fclid=120e6dd7-4668-660d-15c4-7938471767d7&psq=spring+bean+life+cycle&u=a1aHR0cHM6Ly93d3cuZ2Vla3Nmb3JnZWVrcy5vcmcvYmVhbi1saWZlLWN5Y2xlLWluLWphdmEtc3ByaW5nLw&ntb=1).

**Bean Life Cycle Stages**

1. **Instantiation**: The Spring container creates an instance of the bean.
2. **Dependency Injection**: Dependencies are injected into the bean.
3. **Initialization**: Custom initialization methods are called.
4. **Usage**: The bean is ready to be used.
5. **Destruction**: Custom destruction methods are called when the bean is no longer needed.

**Implementing Bean Life Cycle**

**Using XML Configuration**

You can define custom initialization and destruction methods in the Spring XML configuration file.

**Example:**

// HelloWorld.java

package beans;

public class HelloWorld {

public void init() throws Exception {

System.out.println("Bean HelloWorld has been instantiated and I'm the init() method");

}

public void destroy() throws Exception {

System.out.println("Container has been closed and I'm the destroy() method");

}

}

<!-- spring.xml -->

<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN" "http://www.springframework.org/dtd/spring-beans-2.0.dtd">

<beans>

<bean id="hw" class="beans.HelloWorld" init-method="init" destroy-method="destroy"/>

</beans>

// Client.java

package test;

import org.springframework.context.ConfigurableApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Client {

public static void main(String[] args) throws Exception {

ConfigurableApplicationContext cap = new ClassPathXmlApplicationContext("resources/spring.xml");

cap.close();

}

}

**Using Programmatic Approach**

Implement the InitializingBean and DisposableBean interfaces to define custom initialization and destruction methods.

**Example:**

// HelloWorld.java

package beans;

import org.springframework.beans.factory.DisposableBean;

import org.springframework.beans.factory.InitializingBean;

public class HelloWorld implements InitializingBean, DisposableBean {

@Override

public void afterPropertiesSet() throws Exception {

System.out.println("Bean HelloWorld has been instantiated and I'm the init() method");

}

@Override

public void destroy() throws Exception {

System.out.println("Container has been closed and I'm the destroy() method");

}

}

<!-- spring.xml -->

<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN" "http://www.springframework.org/dtd/spring-beans-2.0.dtd">

<beans>

<bean id="hw" class="beans.HelloWorld"/>

</beans>

**Using Annotations**

Annotate the custom initialization and destruction methods with @PostConstruct and @PreDestroy.

**Example:**

// HelloWorld.java

package beans;

import javax.annotation.PostConstruct;

import javax.annotation.PreDestroy;

public class HelloWorld {

@PostConstruct

public void init() throws Exception {

System.out.println("Bean HelloWorld has been instantiated and I'm the init() method");

}

@PreDestroy

public void destroy() throws Exception {

System.out.println("Container has been closed and I'm the destroy() method");

}

}

<!-- spring.xml -->

<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN" "http://www.springframework.org/dtd/spring-beans-2.0.dtd">

<beans>

<bean class="org.springframework.context.annotation.CommonAnnotationBeanPostProcessor"/>

<bean class="beans.HelloWorld"/>

</beans>

**Conclusion**

The Spring framework provides multiple ways to manage the bean life cycle, including XML configuration, programmatic approach, and annotations. Each method offers flexibility and customization options, allowing developers to choose the best approach based on their specific requirements[2](https://www.bing.com/ck/a?!&&p=735aeb113a56049f5d2a13e08abe0e1ea222ee6fcf75f19c993692a8011bce47JmltdHM9MTc0OTQyNzIwMA&ptn=3&ver=2&hsh=4&fclid=120e6dd7-4668-660d-15c4-7938471767d7&u=a1aHR0cHM6Ly9ob3d0b2RvaW5qYXZhLmNvbS9zcHJpbmctY29yZS9zcHJpbmctYmVhbi1saWZlLWN5Y2xlLw&ntb=1)[3](https://www.bing.com/ck/a?!&&p=7b71864c8a7a8f35a8a03ad8e0a3403f37eabaf39ffa73fd7ebd22a032ef61e4JmltdHM9MTc0OTQyNzIwMA&ptn=3&ver=2&hsh=4&fclid=120e6dd7-4668-660d-15c4-7938471767d7&u=a1aHR0cHM6Ly93d3cudHV0b3JpYWxzcG9pbnQuY29tL3NwcmluZy9zcHJpbmdfYmVhbl9saWZlX2N5Y2xlLmh0bQ&ntb=1).