1. ApplicationContext is more preferred way than BeanFactory
2. In new Spring versions BeanFactory is replaced with ApplicationContext. But still BeanFactory exists for backward compatability
3. ApplicationContext extends BeanFactory and has the following benefits
   * it supports internationalization for text messages
   * it supports event publication to the registered listeners
   * access to the resources such as URLs and files
4. **BeanFactory** instantiate bean when you call **getBean()** method while ApplicationContext instantiate Singleton bean when container is started, It doesn't wait for getBean() to be called.
5. **BeanFactory** doesn't provide support for internationalization but **ApplicationContext** provides support for it.
6. Another difference between **BeanFactory** vs **ApplicationContext** is ability to publish event to beans that are registered as listener.
7. One of the popular implementation of **BeanFactory** interface is **XMLBeanFactory** while one of the popular implementation of **ApplicationContext** interface is **ClassPathXmlApplicationContext**.
8. If you are using auto wiring and using **BeanFactory** than you need to register **AutoWiredBeanPostProcessor** using API which you can configure in XML if you are using **ApplicationContext**. In summary **BeanFactory** is OK for testing and non production use but **ApplicationContext** is more feature rich container implementation and should be favored over **BeanFactory**
9. **BeanFactory** by default its support **Lazy** loading and **ApplicationContext** by default support **Aggresive** loading.

BeanFactory :

1. Does not support the Annotation based dependency Injection.
2. Doesn't Support I18N.
3. By default its support Lazy loading.
4. it doesn't allow configure to multiple configuration files.

ex: BeanFactory context=new XmlBeanFactory(new Resource("applicationContext.xml"));

ApplicationContext

1. Support Annotation based dependency Injection.-@Autowired, @PreDestroy
2. Support I18N
3. Its By default support Aggresive loading.
4. It allow to configure multiple configuration files.

|  | **BeanFactory** | **ApplicationContext** |
| --- | --- | --- |
| Annotation support | No | Yes |
| BeanPostProcessor Registration | Manual | Automatic |
| Implementation | XMLBeanFactory | ClassPath/FileSystem/WebXmlApplicationContext |
| Internationalization | No | Yes |
| Enterprise services | No | Yes |
| ApplicationEvent publication | No | Yes |

Loading Lazy loading EagerLoading

**1. What is Life Cycle of a Bean?**

A Spring bean needs to be instantiated when the container starts, based on Java or XML bean definition. The framework may also be required to perform some pre and post-initialization steps to get the bean into a usable state.

After that, when the bean is no longer required, it will be removed from the IoC container. Like the initialization phase, the Spring framework may need to perform pre-and post-destruction steps to free the other system resources.

The [Spring bean factory](https://howtodoinjava.com/spring-core/how-to-create-beans-using-spring-factorybean/) is responsible for managing the life cycle callbacks of the beans which are created in the spring containers.

**1.1. Life Cycle Callback Methods**

Spring bean factory controls the creation and destruction of beans. To execute some custom code, the bean factory provides the callback methods, which can be categorized broadly into two groups:

1. **Post-initialization** callback methods
2. **Pre-destruction** callback methods

Spring Bean Life Cycle

**2. How to Customize the Bean Life Cycle**

Spring framework provides the following **four ways for controlling life cycle events** of a bean:

1. InitializingBean and DisposableBean callback interfaces
2. \*Aware interfaces for specific behavior
3. Custom init() and destroy() methods in bean configuration file
4. @PostConstruct and @PreDestroy annotations

Let’s learn about each way in some detail.

**2.1. InitializingBean and DisposableBean Interfaces**

The [org.springframework.beans.factory.InitializingBean](https://docs.spring.io/spring-framework/docs/3.0.x/javadoc-api/org/springframework/beans/factory/InitializingBean.html) interface allows a bean to perform initialization work after all necessary properties on the bean have been set by the container.

The InitializingBean interface specifies a single method:

InitializingBean.java

**void** afterPropertiesSet() **throws** Exception;

The afterPropertiesSet() method is not a preferable way to initialize the bean because it tightly couples the bean class with the spring container. A better approach is to use “*init-method*” attribute in bean definition in applicationContext.xml.

Similarly, implementing the [org.springframework.beans.factory.DisposableBean](https://docs.spring.io/spring/docs/1.2.9/api/org/springframework/beans/factory/DisposableBean.html) interface allows a bean to get a callback before the Spring container destroys the bean.

The DisposableBean interface specifies a single method:

DisposableBean.java

**void** destroy() **throws** Exception;

A sample bean implementing the above interfaces would look like this:

DemoBean.java

**package** com.howtodoinjava.task;

**import** org.springframework.beans.factory.DisposableBean;

**import** org.springframework.beans.factory.InitializingBean;

**public** **class** DemoBean **implements** InitializingBean, DisposableBean

{

*//Other bean attributes and methods*

@Override

**public** **void** afterPropertiesSet() **throws** Exception

{

*//Bean initialization code*

}

@Override

**public** **void** destroy() **throws** Exception

{

*//Bean destruction code*

}

}

**2.2. \*Aware Interfaces to Add Specific Behavior**

Spring offers a range of interfaces that allow the beans to indicate to the container that they require a particular infrastructure dependency.

Each of these Aware interfaces will require us to implement a method to inject the dependency in the bean.

We can summarize these interfaces as :

|  |  |  |
| --- | --- | --- |
| **Aware interface** | **Method to override** | **Purpose** |
| ApplicationContextAware | void setApplicationContext (ApplicationContext applicationContext) throws BeansException; | Interface to be implemented by any object that wishes to be notified of the ApplicationContext that it runs in. |
| ApplicationEventPublisherAware | void setApplicationEventPublisher (ApplicationEventPublisher applicationEventPublisher); | Set the ApplicationEventPublisher that this object runs in. |
| BeanClassLoaderAware | void setBeanClassLoader (ClassLoader classLoader); | Callback that supplies the bean class loader to a bean instance. |
| BeanFactoryAware | void setBeanFactory (BeanFactory beanFactory) throws BeansException; | Callback that supplies the owning factory to a bean instance. |
| BeanNameAware | void setBeanName(String name); | Set the name of the bean in the bean factory that created this bean. |
| BootstrapContextAware | void setBootstrapContext (BootstrapContext bootstrapContext); | Set the BootstrapContext that this object runs in. |
| LoadTimeWeaverAware | void setLoadTimeWeaver (LoadTimeWeaver loadTimeWeaver); | Set the LoadTimeWeaver of this object’s containing ApplicationContext. |
| MessageSourceAware | void setMessageSource (MessageSource messageSource); | Set the MessageSource that this object runs in. |
| NotificationPublisherAware | void setNotificationPublisher (NotificationPublisher notificationPublisher); | Set the NotificationPublisher instance for the current managed resource instance. |
| PortletConfigAware | void setPortletConfig (PortletConfig portletConfig); | Set the PortletConfig this object runs in. |
| PortletContextAware | void setPortletContext (PortletContext portletContext); | Set the PortletContext that this object runs in. |
| ResourceLoaderAware | void setResourceLoader (ResourceLoader resourceLoader); | Set the ResourceLoader that this object runs in. |
| ServletConfigAware | void setServletConfig (ServletConfig servletConfig); | Set the ServletConfig that this object runs in. |
| ServletContextAware | void setServletContext (ServletContext servletContext); | Set the ServletContext that this object runs in. |

Java program to show the use of Aware interfaces.

DemoBean.java

**package** com.howtodoinjava.task;

**import** org.springframework.beans.BeansException;

**import** org.springframework.context.ApplicationContext;

**import** org.springframework.context.ApplicationContextAware;

**public** **class** DemoBean **implements** ApplicationContextAware

{

**private** ApplicationContext ctx;

@Override

**public** **void** setApplicationContext(ApplicationContext ctx)

**throws** BeansException {

**this**.ctx = ctx;

}

*//Use the context in other bean methods*

}

**2.3. Custom init() and destroy() Methods**

We can add the default init() and destroy() methods in two ways:

* **Local definition**s applicable to a single bean
* **Global definition**s applicable to all beans defined in whole beans context

**2.3.1. Local definitions**

The local init() and destroy() methods are configured as in the given example.

applicationContext.xml

**<beans>**

**<bean id**="demoBean" **class**="com.howtodoinjava.task.DemoBean"

**init-method**="customInit"

**destroy-method**="customDestroy"**></bean>**

**</beans>**

**2.3.2. Global definitions**

The container will invoke the global methods for all bean definitions given under <beans> tag. Global overrides are helpful when we have a pattern of defining common method names such as init() and destroy() for all the beans consistently.

This feature helps us in not mentioning the init and destroy method names for all beans independently.

applicationContext.xml

**<beans default-init-method**="customInit" **default-destroy-method**="customDestroy"**>**

**<bean id**="demoBean" **class**="com.howtodoinjava.task.DemoBean"**></bean>**

**</beans>**

Based on the above local or global overrides, we must write the customInit() and customDestroy() methods in the bean classes as below example.

DemoBean.java

**package** com.howtodoinjava.task;

**public** **class** DemoBean

{

**public** **void** customInit()

{

System.out.println("Method customInit() invoked...");

}

**public** **void** customDestroy()

{

System.out.println("Method customDestroy() invoked...");

}

}

**2.4. @PostConstruct and @PreDestroy Annotations**

From Spring 2.5 onwards, we can use the @PostConstruct and @PreDestroy annotations for specifying the bean life cycle methods.

* **@PostConstruct** annotated method will be *invoked after the bean has been constructed using default constructor* and just before it’s instance is returned to requesting object.
* **@PreDestroy** annotated method is *invoked just before the bean is about be destroyed* inside bean container.

Java program to show usage of **annotation configuration** to control using annotations.

DemoBean.java

**package** com.howtodoinjava.task;

**import** javax.annotation.PostConstruct;

**import** javax.annotation.PreDestroy;

**public** **class** DemoBean

{

@PostConstruct

**public** **void** customInit()

{

System.out.println("Method customInit() invoked...");

}

@PreDestroy

**public** **void** customDestroy()

{

System.out.println("Method customDestroy() invoked...");

}

}