The **Inversion of Control** is a process by which application defines the dependency and **Spring IoC Container** manages these processes as well life cycle of beans. These dependencies are then satisfied in **runtime**by the Spring Framework. The **IoC**container is the main component of the Spring framework. It provides the main **IoC**container and **AOP**framework. The core container of the Spring Framework provides important functionality including [**dependency injection**](https://www.dineshonjava.com/dependency-injection-in-spring/) and **bean life cycle**management.

The IoC container is responsible to instantiate, configure and assemble the objects. The IoC container gets informations from the XML file and works accordingly. The main tasks performed by IoC container are:

* to instantiate the application class
* to configure the object
* to assemble the dependencies between the objects

There are two types of IoC containers. They are:

1. **BeanFactory**
2. **ApplicationContext**

Difference between BeanFactory and the ApplicationContext

The org.springframework.beans.factory.**BeanFactory** and the org.springframework.context.**ApplicationContext** interfaces acts as the IoC container. The ApplicationContext interface is built on top of the BeanFactory interface. It adds some extra functionality than BeanFactory such as simple integration with Spring's AOP, message resource handling (for I18N), event propagation, application layer specific context (e.g. WebApplicationContext) for web application. So it is better to use ApplicationContext than BeanFactory.

## Spring IoC Container

The Spring IoC Container actually comes with two distinct containers:  
**1.**[**Bean Factories**](https://www.dineshonjava.com/what-is-bean-factory-in-spring/)  
**2.**[**Application Context**](https://www.dineshonjava.com/application-context-in-spring/)

### [Bean Factories](https://www.dineshonjava.com/what-is-bean-factory-in-spring/)

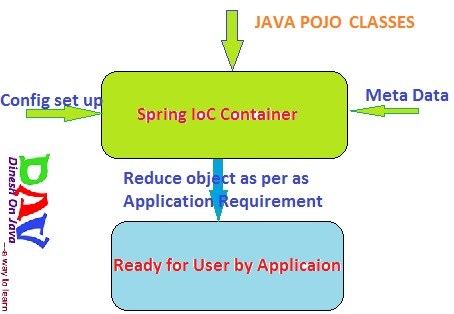
The **[BeanFactory](https://www.dineshonjava.com/what-is-bean-factory-in-spring/)**provides the configuration framework and basic functionality and support for [**Dependency Injection**](https://www.dineshonjava.com/dependency-injection-in-spring/). The **org.springframework.beans.factory.BeanFactory** is the actual representation of the Spring **IoC***container* that is responsible for containing and otherwise managing the aforementioned beans.

The **BeanFactory**interface is the central **IoC**container interface in Spring. Its responsibilities include instantiating or sourcing application objects, configuring such objects, and assembling the dependencies between these objects.

A Spring BeanFactory is like a factory class that contains a collection of beans. The Spring BeanFactory holds Bean Definitions of multiple beans within itself and then instantiates the bean whenever asked for by clients.

* Spring ***BeanFactory***is able to create associations between collaborating objects as they are instantiated. This removes the burden of configuration from bean itself and the beans client.
* Spring ***BeanFactory***also takes part in the life cycle of a bean, making calls to custom initialization and destruction methods.

This is the simplest container providing basic support for **DI** and defined by the **org.springframework.beans.factory.BeanFactory** interface. The ***BeanFactory***and related interfaces, such as ***BeanFactoryAware***, ***InitializingBean***, ***DisposableBean***, are still present in Spring for the purposes of backward compatibility with the large number of third-party frameworks that integrate with Spring.



### [Spring ApplicationContext Container](https://www.dineshonjava.com/application-context-in-spring/):

This container adds more enterprise-specific functionality such as the ability to resolve textual messages from a properties file and the ability to publish application events to interested event listeners. This container is defined by the **org.springframework.context.ApplicationContext interface**.

 we used [Bean Factory](https://www.dineshonjava.com/what-is-bean-factory-in-spring/) container and discussed with example. Now we will discuss about the ApplicationContext and Using with in Example. ApplicationContext like [Bean Factory](https://www.dineshonjava.com/what-is-bean-factory-in-spring/)‘s Big Brother with some additional functionality such as **[AOP](https://www.dineshonjava.com/introduction-to-aop-in-spring/)concept**, **event notification** and it adds more enterprise-specific functionality such as the ability to resolve **textual messages from a properties file** and the ability to **publish application events to interested event listeners**. This container is defined by the **org.springframework.context.ApplicationContext** interface.

## ApplicationContext Implementations

The most commonly used **ApplicationContext** implementations are:

### FileSystemXmlApplicationContext

This container loads the definitions of the beans from an XML file. Here you need to provide the**full path of the XML bean configuration file** to the constructor.

FileSystemXmlApplicationContext context = new FileSystemXmlApplicationContext("F:/my workspace/springAppDemo/src/spring.xml");

**Using wildcard for file system:**

FileSystemXmlApplicationContext context = new FileSystemXmlApplicationContext("F:/\*my workspace\*/\*\*/src/\*-spring.xml");

### ClassPathXmlApplicationContext

This container loads the definitions of the beans from an XML file. Here you do not need to provide the full path of the XML file but you need to set **CLASSPATH** properly because this container will look bean configuration XML file in CLASSPATH.

ApplicationContext context = new ClassPathXmlApplicationContext("classpath:com/dineshonjava/sdnext/springConfig/spring.xml");

**Using wildcard for claspath:**

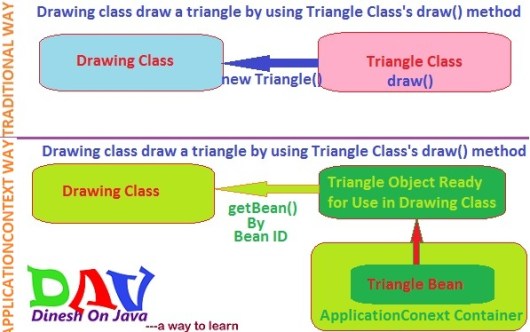
ApplicationContext context = new ClassPathXmlApplicationContext("classpath\*:com/dineshonjava/\*\*/springConfig/spring.xml");

ApplicationContext context = new ClassPathXmlApplicationContext("classpath\*:com/\*/\*\*/springConfig/\*-spring.xml");

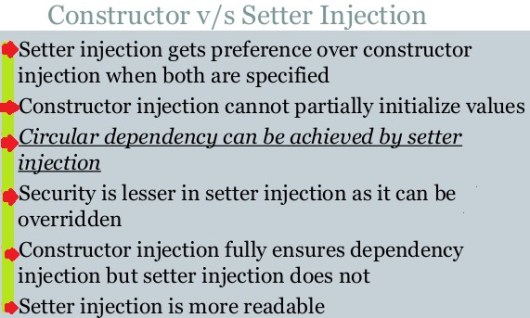
### WebXmlApplicationContext

This container loads the XML file with definitions of all beans from within a web application.

ApplicationContext appContext = WebApplicationContextUtils.getWebApplicationContext(servletContext);



These are two ways to define the dependency injection in the spring application. Spring Dependency Injection (DI) design pattern is used to define the object dependencies between each other. There are following two types in [**dependency-injection**](https://www.dineshonjava.com/dependency-injection-in-spring/):



Bean **Autowiring**reduces the effort of writing **properties or constructor arguments**. Bean **Autowiring**is the feature provided by the spring framework to skip the some of configuration in XML file. The bean **autowiring**in specified at the **autowire**attribute inside <bean></bean> element.

Autowiring feature of spring framework enables you to inject the object dependency implicitly. It internally uses setter or constructor injection.

Autowiring can't be used to inject primitive and string values. It works with reference only.

In the Spring Framework bean model, beans have relationships with other beans. In order to satisfy these dependencies of injecting one bean into another we use autowire functionality of Spring Framework. In this tutorial we are taking 6 scenarios by which Spring autowires one bean to another. The Scenarios are changed based on Spring Configuration files and rest all the source code given in this tutorial is the same.

In the above scenario we are using autowire attribute value as "no". It means we are using no auto-wiring, so we have to provide ref attribute for referring any bean.

In the above scenario we are using autowire attribute value as "byName". It means we are using auto-wiring based on name of property, so here Employee has a Address property. Here we have a property in Employee by name as : Address address. So when we say that auto-wire by name Spring sees in configuration that whether there is any bean with id equal to name of property. As soon as it finds this property and its matching id, it uses its setter method to set the property .

n the above scenario we are using autowire attribute value as "byType". It means we are using auto-wiring based on type of property, so here Employee has a Address property. Here we have a property in Employee by type as : com.hubberspot.spring.autowire.Address . So when we say that auto-wire by type Spring sees in configuration that whether there is any bean with class equal to type of property. As soon as it finds any matching, it uses its setter method to set the property. In case of byType autowiring mode, bean id and reference name may be different. But there must be only one bean of a type. In this case, it works fine because you have created an instance of B type. It doesn't matter that you have different bean name than reference name.

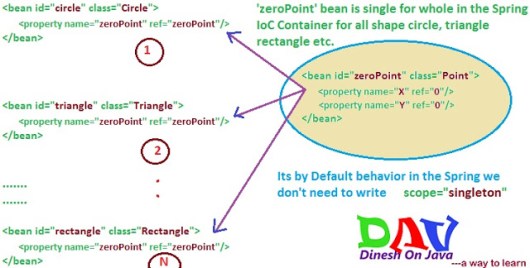
But, if you have multiple bean of one type, it will not work and throw exception.

In the above scenario we are using autowire attribute value as "constructor". It means we are using auto-wiring based on constructor arguments , so here Employee has a Address property. Here we have a constructor in Employee class which takes a argument as : Address address. So when we say that auto-wire by constructor Spring sees in configuration that whether there is any bean with id equal to constructor argument. As soon as it finds any matching , it uses its setter method to set the property .

When a bean is a **singleton**, only one *shared* instance of the bean will be managed, and all requests for beans with an***id or ids***matching that bean definition will result in that one specific bean instance being returned by the [Spring container](https://www.dineshonjava.com/2012/06/spring-ioc-container.html).

We can say another way, **when you define a bean definition and it is scoped as a singleton,** then the [Spring IoC container](https://www.dineshonjava.com/2012/06/spring-ioc-container.html) will create ***exactly one* *instance***of the object defined by that bean definition. This single instance will be stored in a cache of such singleton beans, and ***all***

**NOTE** : This singleton is differ from the singleton pattern in Java Class. **Single pattern in java mean you can create the only one instance of a that class in**[**JVM**](https://www.dineshonjava.com/java-virtual-machine/). **But In spring singleton bean scope means every container can create only single bean in the**[**Spring IoC Container**](https://www.dineshonjava.com/spring-ioc-container/)**but a**[**JVM**](https://www.dineshonjava.com/java-virtual-machine/)**can have multiple**[**Spring IoC Container**](https://www.dineshonjava.com/spring-ioc-container/)**so**[**JVM**](https://www.dineshonjava.com/java-virtual-machine/)**can multiple beans rather than bean singleton bean scope.**



**Prototype Bean Scope:**If scope is set to prototype, the [Spring IoC container](https://www.dineshonjava.com/2012/06/spring-ioc-container.html) creates new bean instance of the object every time a request for that specific bean is made. **As a rule, use the prototype scope for all state-full beans and the singleton scope for stateless beans.**

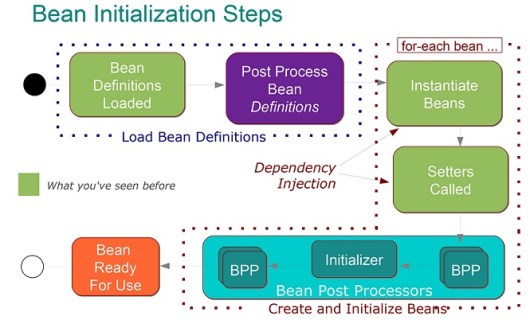
Spring supports bean configuration inheritance. We can define a bean and then further configure it to create new bean.

By using the bean inheritance we can share common values, properties or configurations.

A child bean inherits its parent bean configurations, properties and attributes.

In addition, the child beans can override the inherited value.

A spring bean definition contains lot of information like property values, constructor arguments, and container specific information like init and destroy method settings and so on. Spring allows to inherit all these bean properties. A child bean definition can inherit configurations from its parent definition. The child bean definition can override some values, or add new values.



The Spring Framework provides several marker interfaces to change the behavior of your bean in the [container](https://www.dineshonjava.com/spring-ioc-container/); they include***InitializingBean***and ***DisposableBean***. Implementing these interfaces will result in the container calling ***afterPropertiesSet***() for the former and ***destroy***() for the latter to allow the bean to perform certain actions upon initialization and destruction.

When a bean is instantiated, it may be required to perform some initialization to get it into a usable state. Similarly, when the bean is no longer required and is removed from the [container](https://www.dineshonjava.com/spring-ioc-container/), some cleanup may be required.

Though, there is lists of the activities that take place behind the scenes between the time of bean Instantiation and its destruction, but this chapter will discuss only **two important bean life cycle callback methods** which are required at the time of bean initialization and its destruction.

Beans can be notified after creation and all properties are set, and before they are destroyed and removed from the [bean container](https://www.dineshonjava.com/spring-ioc-container/). This involves specifying the callback method to be invoked by the [container](https://www.dineshonjava.com/spring-ioc-container/). This is done in XML by specifying attributes ***init-method=”myinit”***, for the initialization callback, and ***destroy-method=”mydestroy”,*** for the destroy callback. “***myinit***” and “***cleanUp***” are names of instance methods in the bean class.

### Initialization callbacks

Implementing the org.springframework.beans.factory.InitializingBean interface allows a bean to perform initialization work after all necessary properties on the bean are set by the container. The InitializingBean interface specifies exactly one method:  
org.springframework.beans.factory.InitializingBean interface  provide Initialization callbacks method as given below..

void afterPropertiesSet() throws Exception

Now we can implements above interface and do some initialization functionality with in this method. As below..

public class Triangle implements InitializingBean

{

@Override

public void afterPropertiesSet() throws Exception

{

//To do some initialization works here

System.out.println("InitializingBean init method is called for Triangle");

}

}

Generally, the use of the InitializingBean interface can be avoided (and is discouraged since it unnecessarily couples the code to Spring). A bean definition provides support for a generic initialization method to be specified. In the case of XML-based configuration metadata, this is done using the ‘init-method’ attribute. For example, the following definition:  
In the case of **XML-based configuration** metadata, we can use the **init-method** attribute to specify the name of the method that has a void no-argument signature. For example:

<bean class="com.dineshonjava.sdnext.callbackLifecycle.tutorial.Triangle" id="triangle" init-method="myInit"></bean>

Now following has **myInit**method in class.

public class Triangle

{

public void myInit()

{

//To do some initialization works here

System.out.println("My init method is called for Triangle");

}

}

**Now using Java annotations can also be used to declare bean life cycle callbacks.**

public class Triangle

{

//init callback

@PostConstruct

public void myInit()

{

//To do some initialization works here

System.out.println("My init method is called for Triangle");

}

}

### Destruction callbacks

Implementing the org.springframework.beans.factory.DisposableBean interface allows a bean to get a callback when the container containing it is destroyed. The DisposableBean interface specifies one method:

void destroy() throws Exception

Now we can implements above interface and do some Destruction functionality with in this method. As below..

public class Triangle implements DisposableBean

{

@Override

public void destroy() throws Exception

{

//To do some Destruction works here

System.out.println("DisposableBean destroy method is called for Triangle");

}

}

Generally, the use of the DisposableBean marker interface can be avoided (and is discouraged since it unnecessarily couples the code to Spring). A bean definition provides support for a generic destroy method to be specified. When using XML-based configuration metadata this is done via the ‘destroy-method’ attribute on the . For example, the following definition:  
In the case of **XML-based configuration** metadata, we can use the **destroy-method** attribute to specify the name of the method that has a void no-argument signature. For example:

<bean class="com.dineshonjava.sdnext.callbackLifecycle.tutorial.Triangle" destroy-method="cleanUp" id="triangle"></bean>

Now following has **cleanUp**method in class.

public class Triangle

{

public void cleanUp()

{

//To do some Destruction works here

System.out.println("cleanUp method is called for Triangle");

}

}

**Now using Java annotations can also be used to declare bean life cycle callbacks.**

public class Triangle

{

//destroy callback

@PreDestroy

public void myInit()

{

//To do some Destruction works here

System.out.println("cleanUp method is called for Triangle");

}

}

If you are using [**Spring’s IoC container**](https://www.dineshonjava.com/spring-ioc-container/) in a non-web application environment; for example, in a rich client desktop environment; you register a shutdown hook with the [**JVM**](https://www.dineshonjava.com/java-virtual-machine/). Doing so ensures a graceful shutdown and calls the relevant destroy methods on your singleton beans so that all resources are released.

It is recommended that you do not use the InitializingBean or DisposableBean callbacks, because XML configuration gives much flexibility in terms of naming your method.

BeanPostProcessors interface provides methods that you can implement to have your own instantiation logic.Also you can write your own logic after spring IOC finishes instantiating, configuring, and initializing a bean by plugging in one or more BeanPostProcessor implementations.

You can configure multiple BeanPostProcessors and also can decide the order in which they will run relative to each other by setting order property but foe that BeanPostProcessors  have to implement ordered interface.

xtension of BeanPostProcessor is BeanFactoryPostProcessor interface which allows direct modification of bean definitions before a bean is instantiated

An ApplicationContext will automatically register and process a bean that implements either of these interfaces, but a BeanFactory would have to have a BeanPostProcessor or BeanFactoryPostProcessor registered with it programatically.

BeanPostProcessor is used to interact with newly created bean instances *before* and/or *after* their initialization method is invoked by the Spring container. You can use BeanPostProcessor to execute custom logic *before* and/or *after* bean’s initialization method is invoked by the Spring container. Once the BeanPostProcessor beans are created, the Spring container invokes each BeanPostProcessor’s postProcessBeforeInitialization and postProcessAfterInitialization methods for each bean instance created by the Spring container.

### BeanPostProcessor example – Validating bean instances

In a Spring application, you may want to verify that a bean instance is configured correctly before it is injected into dependent beans or accessed by other objects in the application. Let’s see how we can use a BeanPostProcessor implementation to give an opportunity to each bean instance to validate its configuration before the bean instance is made available to dependent beans or other application objects.

The following example listing shows an InstanceValidator interface that must be implemented by beans whose configuration we want to validate using a BeanPostProcessor implementation:

**Example listing**– InstanceValidator interface

package sample.spring.chapter04.springbankapp.common;

public interface InstanceValidator {

  void validateInstance();

}

InstanceValidator interface defines a validateInstance method that verifies whether the bean instance was correctly initialized or not. We’ll soon see that the validateInstance method is invoked by a BeanPostProcessor implementation.

The following example listing shows the FixedDepositDaoImpl class that implements InstanceValidator interface:

**Example listing** – FixedDepositDaoImpl class

package sample.spring.chapter04.springbankapp.dao;

import org.apache.log4j.Logger;

import sample.spring.chapter04.springbankapp.common.InstanceValidator;

public class FixedDepositDaoImpl implements FixedDepositDao, InstanceValidator {

  private static Logger logger = Logger.getLogger(FixedDepositDaoImpl.class);

  private DatabaseConnection connection;

  public FixedDepositDaoImpl() {

    logger.info("FixedDepositDaoImpl's constructor invoked");

  }

  public void initializeDbConnection() {

  logger.info("FixedDepositDaoImpl's initializeDbConnection method invoked");

   connection = DatabaseConnection.getInstance();

  }

  @Override

  public void validateInstance() {

    logger.info("Validating FixedDepositDaoImpl instance");

     if(connection == null) {

     logger.error("Failed to obtain DatabaseConnection instance");

   }

  }

}

Multiple lifecycle mechanisms configured for the same bean, with different initialization methods, are called as follows:

* Methods annotated with @PostConstruct
* afterPropertiesSet() as defined by the InitializingBean callback interface
* A custom configured init() method

Destroy methods are called in the same order:

* Methods annotated with @PreDestroy
* destroy() as defined by the DisposableBean callback interface
* A custom configured destroy() method

***BeanPostProcessor***interface has two method…

* 1. ***postProcessAfterInitialization(Object bean, String beanName)*** execute after initialization of each beans in the [**Spring IoC Container**](https://www.dineshonjava.com/2012/06/spring-ioc-container.html).
* 2. ***postProcessBeforeInitialization(Object bean, String beanName)*** execute before initialization of each beans in the [**Spring IoC Container**](https://www.dineshonjava.com/2012/06/spring-ioc-container.html).

Sometimes we want to initialize resources in the bean classes, for example creating database connections or validating third party services at the time of initialization before any client request. [Spring framework](https://www.journaldev.com/16922/spring-framework)provide different ways through which we can provide post-initialization and pre-destroy methods in a spring bean life cycle.

1. By implementing **InitializingBean** and **DisposableBean** interfaces – Both these interfaces declare a single method where we can initialize/close resources in the bean. For post-initialization, we can implement InitializingBean interface and provide implementation of afterPropertiesSet()method. For pre-destroy, we can implement DisposableBean interface and provide implementation of destroy() method. These methods are the callback methods and similar to servlet listener implementations.

This approach is simple to use but it’s not recommended because it will create tight coupling with the Spring framework in our bean implementations.

1. Providing **init-method** and **destroy-method** attribute values for the bean in the spring bean configuration file. This is the recommended approach because of no direct dependency to spring framework and we can create our own methods.

Note that both *post-init* and *pre-destroy* methods should have no arguments but they can throw Exceptions. We would also require to get the bean instance from the spring application context for these methods invocation.