Spring:

### Spring Bean

Spring Bean is nothing special, any object in the Spring framework that we initialize through Spring container is called Spring Bean. Any normal Java POJO class can be a Spring Bean if it’s configured to be initialized via container by providing configuration metadata information.

Spring **ApplicationContext** is responsible to initialize the Spring Beans defined in spring bean configuration fil

Spring is a *lightweight* framework. It can be thought of as a *framework of frameworks* because it provides support to various frameworks such as Struts, Hibernate, Tapestry, EJB, JSF etc. The framework, in broader sense, can be defined as a structure where we find solution of the various technical problems.

The Spring framework comprises several modules such as IOC, AOP, DAO, Context, ORM, WEB MVC etc. We will learn these modules in next page. Let's understand the IOC and Dependency Injection first.

Thus, IOC makes the code loosely coupled. In such case, there is no need to modify the code if our logic is moved to new environment.

In Spring framework, IOC container is responsible to inject the dependency. We provide metadata to the IOC container either by XML file or annotation.

Advantage of Dependency Injection

* makes the code loosely coupled so easy to maintain
* makes the code easy to test

Advantages of Spring Framework

There are many advantages of Spring Framework. They are as follows:

1) Predefined Templates

Spring framework provides templates for JDBC, Hibernate, JPA etc. technologies. So there is no need to write too much code. It hides the basic steps of these technologies.

Let's take the example of JdbcTemplate, you don't need to write the code for exception handling, creating connection, creating statement, committing transaction, closing connection etc. You need to write the code of executing query only. Thus, it save a lot of JDBC code.

2) Loose Coupling

The Spring applications are loosely coupled because of dependency injection.

3) Easy to test

The Dependency Injection makes easier to test the application. The EJB or Struts application require server to run the application but Spring framework doesn't require server.

4) Lightweight

Spring framework is lightweight because of its POJO implementation. The Spring Framework doesn't force the programmer to inherit any class or implement any interface. That is why it is said non-invasive.

5) Fast Development

The Dependency Injection feature of Spring Framework and it support to various frameworks makes the easy development of JavaEE application.

6) Powerful abstraction

It provides powerful abstraction to JavaEE specifications such as JMS, JDBC, JPA and JTA.

7) Declarative support

It provides declarative support for caching, validation, transactions and formatting.

Beans:-

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. These beans are created with the configuration metadata that you supply to the container. For example, in the form of XML <bean/> definitions which you have already seen in the previous chapters.

Bean definition contains the information called **configuration metadata**, which is needed for the container to know the following −

* How to create a bean
* Bean's lifecycle details
* Bean's dependencies

All the above configuration metadata translates into a set of the following properties that make up each bean definition.

|  |  |
| --- | --- |
| **Sr.No.** | **Properties & Description** |
| 1 | **class**  This attribute is mandatory and specifies the bean class to be used to create the bean. |
| 2 | **name**  This attribute specifies the bean identifier uniquely. In XMLbased configuration metadata, you use the id and/or name attributes to specify the bean identifier(s). |
| 3 | **scope**  This attribute specifies the scope of the objects created from a particular bean definition and it will be discussed in bean scopes chapter. |
| 4 | **constructor-arg**  This is used to inject the dependencies and will be discussed in subsequent chapters. |
| 5 | **properties**  This is used to inject the dependencies and will be discussed in subsequent chapters. |
| 6 | **autowiring mode**  This is used to inject the dependencies and will be discussed in subsequent chapters. |
| 7 | **lazy-initialization mode**  A lazy-initialized bean tells the IoC container to create a bean instance when it is first requested, rather than at the startup. |
| 8 | **initialization method**  A callback to be called just after all necessary properties on the bean have been set by the container. It will be discussed in bean life cycle chapter. |
| 9 | **destruction method**  A callback to be used when the container containing the bean is destroyed. It will be discussed in bean life cycle chapter. |

## Spring Configuration Metadata

Spring IoC container is totally decoupled from the format in which this configuration metadata is actually written. Following are the three important methods to provide configuration metadata to the Spring Container −

* XML based configuration file.
* Annotation-based configuration
* Java-based configuration

You already have seen how XML-based configuration metadata is provided to the container, but let us see another sample of XML-based configuration file with different bean definitions including lazy initialization, initialization method, and destruction method −

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {

public static void main(String[] args) {

ApplicationContext context = new ClassPathXmlApplicationContext("Beans.xml");

HelloWorld obj = (HelloWorld) context.getBean("helloWorld");

obj.getMessage();

}

* The first step is to create an application context where we used framework API **ClassPathXmlApplicationContext()**. This API loads beans configuration file and eventually based on the provided API, it takes care of creating and initializing all the objects, i.e. beans mentioned in the configuration file.
* The second step is used to get the required bean using **getBean()**method of the created context. This method uses bean ID to return a generic object, which finally can be casted to the actual object. Once you have an object, you can use this object to call any class method.
* <?xml version = *"1.0"* encoding = *"UTF-8"*?>
* <beans xmlns = *"http://www.springframework.org/schema/beans"*
* xmlns:xsi = *"http://www.w3.org/2001/XMLSchema-instance"*
* xsi:schemaLocation = *"http://www.springframework.org/schema/beans*
* *http://www.springframework.org/schema/beans/spring-beans-3.0.xsd"*>
* <bean id = *"helloWorld"* class = *"com.tutorialspoint.HelloWorld"*>
* <property name = *"message"* value = *"Hello World!"*/>
* </bean>
* </beans>
* Usually developers name this file as **Beans.xml**, but you are independent to choose any name you like. You have to make sure that this file is available in CLASSPATH and use the same name in the main application while creating an application context as shown in MainApp.java file.
* The Beans.xml is used to assign unique IDs to different beans and to control the creation of objects with different values without impacting any of the Spring source files. For example, using the following file you can pass any value for "message" variable and you can print different values of message without impacting HelloWorld.java and MainApp.java files. Let us see how it works −
* The Spring container is at the core of the Spring Framework. The container will create the objects, wire them together, configure them, and manage their complete life cycle from creation till destruction. The Spring container uses DI to manage the components that make up an application. These objects are called Spring Beans, which we will discuss in the next chapter.
* The container gets its instructions on what objects to instantiate, configure, and assemble by reading the configuration metadata provided. The configuration metadata can be represented either by XML, Java annotations, or Java code. The following diagram represents a high-level view of how Spring works. The Spring IoC container makes use of Java POJO classes and configuration metadata to produce a fully configured and executable system or application.
* 

## Other:- Both BeanFactory and ApplicationContext provides a way to get a bean from Spring IOC container by calling getBean("bean name"), but there is some difference in there working and features provided by them. One difference between bean factory and application context is that former only instantiate bean when you call getBean() method while ApplicationContext instantiates Singleton bean when the container is started,  It doesn't wait for getBean to be called. BeanFactory vs ApplicationContext in Spring

Before seeing difference between ApplicationContext and BeanFactory, let see some similarity between both of them. Spring provides two kinds of IOC container, one is BeanFactory and other is ApplicationContext. Syntactically BeanFactory and ApplicationContext both are [Java interfaces](http://javarevisited.blogspot.in/2012/04/10-points-on-interface-in-java-with.html) and ApplicationContext extends BeanFactory. Both of them are configuration using [XML configuration file](http://javarevisited.blogspot.in/2012/03/how-to-read-properties-file-in-java-xml.html). In short BeanFactory provides basic IOC and DI features while ApplicationContext provides advanced features. Apart from these, Here are few more difference between BeanFactory and ApplicationContext which is mostly based upon features supported by them.

1) BeanFactory doesn't provide support for internationalization i.e. i18n but ApplicationContext provides support for it.

2) Another difference between BeanFactory vs ApplicationContext is ability to publish event to beans that are registered as listener.

3) One of the popular implementation of BeanFactory interface is XMLBeanFactory while one of the popular implementation of ApplicationContext interface is ClassPathXmlApplicationContext. On [Java web application](http://javarevisited.blogspot.sg/2012/08/what-is-jsessionid-in-j2ee-web.html) we use WebApplicationContext  which extends ApplicationContext interface and adds getServletContext method.

4) 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](http://javarevisited.blogspot.in/2011/09/how-to-write-production-quality-code.html) use but ApplicationContext is more feature rich container implementation and should be favored over BeanFactory

[](http://2.bp.blogspot.com/-jlIT7z2RuQg/T9whph8c-FI/AAAAAAAAAYw/TrAvT017-1c/s1600/spring_thumbnail.PNG)These were some worth noting difference between BeanFactory and ApplicationContext in Spring framework. In most practical cases you will be using ApplicationContext but knowing about BeanFactory is important to understand fundamental concept of spring framework. I mostly use XML configuration file and ClassPathXmlApplicationContext to quickly run any Spring based Java program from [Eclipse](http://javarevisited.blogspot.sg/2012/10/eclipse-shortcut-to-remove-all-unused-imports-java.html) by using following snippet of code :

**public** **static** **void** main(**String** args[]){  
    ApplicationContext ctx =**new** ClassPathXmlApplicationContext("beans.xml");  
    Hello hello =(Hello) ctx.getBean("hello");  
    hello.sayHello("John");  
}

here beans.xml is your spring configuration file and “hello” is a bean defined in that spring configuration file. Here we have used ClassPathXmlApplicationContext  which is an implementation of ApplicationContext interface in Spring.

Other: after Spring beans are created, they need to perform some additional initialization operations so that they can carry out their responsibilities. e.g. When an ATM (Automated Teller Machine) system is started, it needs to connect to its bank’s network.  
  
Similary, sometimes Spring beans need to peform some cleanup operations before it gets destroyed. e.g.when an ATM system is closed it needs to disconnect from its bank network.

Additional initialization can be done by implementing the  
*InitializingBean*  
interface and overriding the  
*afterPropertiesSet()*  
method.  
  
Similarly, cleanup can be done by implementing the  
*DisposableBean*  
interface and overriding the  
*destroy()*  
method.

The drawback of using this approach is that our classes will be required to use the Spring API. If you want to use the class outside the Spring container using the other approach mentioned above is a better way to go.

void **afterPropertiesSet**()

throws [Exception](http://java.sun.com/javase/6/docs/api/java/lang/Exception.html?is-external=true)

Invoked by a BeanFactory after it has set all bean properties supplied (and satisfied BeanFactoryAware and ApplicationContextAware).

This method allows the bean instance to perform initialization only possible when all bean properties have been set and to throw an exception in the event of misconfiguration.

**Throws:**

[Exception](http://java.sun.com/javase/6/docs/api/java/lang/Exception.html?is-external=true) - in the event of misconfiguration (such as failure to set an essential property) or if initialization fails.

**destroy**

void **destroy**()

throws [Exception](http://java.sun.com/javase/6/docs/api/java/lang/Exception.html?is-external=true)

Invoked by a BeanFactory on destruction of a singleton.

**Throws:**

[Exception](http://java.sun.com/javase/6/docs/api/java/lang/Exception.html?is-external=true) - in case of shutdown errors. Exceptions will get logged but not rethrown to allow other beans to release their resources too.

 http://www.springframework.org/schema/beans/spring-beans-3.0.xsd"

    default-init-method="init"

    default-destroy-method="destroy">

<bean id="customerService" class="com.mkyong.customer.services.CustomerService"

init-method="initIt" destroy-method="cleanUp">

## The singleton scope

If a scope is set to singleton, the Spring IoC container creates exactly one instance of the object defined by that bean definition. This single instance is stored in a cache of such singleton beans, and all subsequent requests and references for that named bean return the cached object.

The default scope is always singleton. However, when you need one and only one instance of a bean

Singleton scoped beans are mostly used for stateless beans.

## The prototype scope

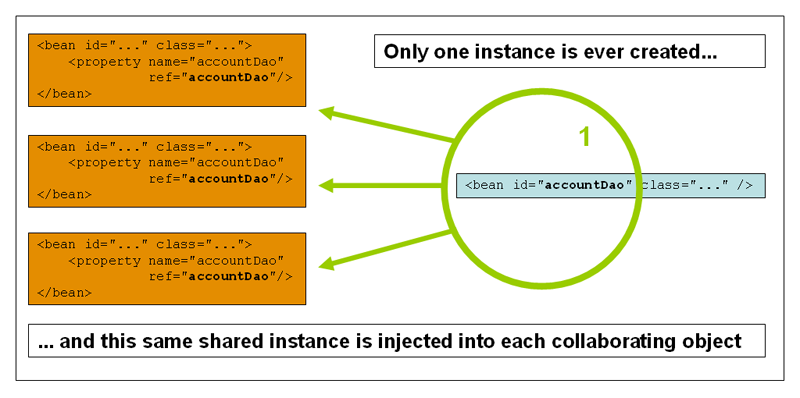
If the scope is set to prototype, the Spring IoC container creates a new bean instance of the object every time a request for that specific bean is made. As a rule, use the prototype scope for all state-full beans and the singleton scope for stateless beans.

* **singleton** (Default) Single bean definition for a single object instance per container.
* **prototype** Single bean definition for multiple instances.
* **request** Single bean definition for a single HTTP request.
* **session** Single bean definition for the HTTP session.
* **global-session** Single bean definition for the global HTTP session. Typically only valid when used in portlet context.
* **application** Single bean definition for the entire application bound to the lifecycle of a ServletContext

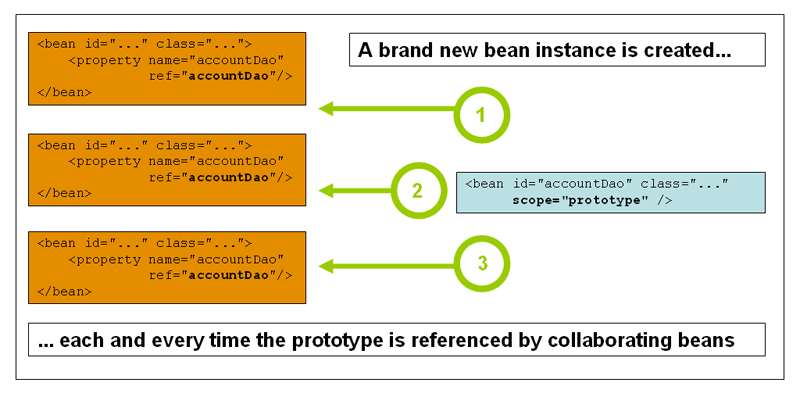
Request, session and application are only valid in the context of a web-aware ApplicationContext

## Singleton and Prototype

**The Singleton scopes the bean definition to a single instance per Spring IoC container (default)**. If scope is set to singleton, the Spring IoC container creates exactly one instance of the object defined by that bean definition. This single instance is stored in a cache of such singleton beans, and all subsequent requests and references for that named bean return the cached object. You have to carefully understand that it is single for its own IoC container, not the JVM or your entire application. Because your application may have more than one IoC container.

[](https://javabeat.net/wp-content/uploads/2013/03/singleton.png)

The **Prototype** scopes a single bean definition to have any number of object instances. If scope is set to prototype, the Spring IoC container 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.**

[](https://javabeat.net/wp-content/uploads/2013/03/prototype.png)

#### 1.5.3. Singleton beans with prototype-bean dependencies

When you use singleton-scoped beans with dependencies on prototype beans, be aware that dependencies are resolved at instantiation time. Thus if you dependency-inject a prototype-scoped bean into a singleton-scoped bean, a new prototype bean is instantiated and then dependency-injected into the singleton bean. The prototype instance is the sole instance that is ever supplied to the singleton-scoped bean.

However, suppose you want the singleton-scoped bean to acquire a new instance of the prototype-scoped bean repeatedly at runtime. You cannot dependency-inject a prototype-scoped bean into your singleton bean, because that injection occurs onlyonce, when the Spring container is instantiating the singleton bean and resolving and injecting its dependencies. If you need a new instance of a prototype bean at runtime more than once, see [Method injection](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-method-injection)

LifeCycle:

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, some cleanup may be required.

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

To define setup and teardown for a bean, we simply declare the <bean> with **initmethod** and/or **destroy-method** parameters. The init-method attribute specifies a method that is to be called on the bean immediately upon instantiation. Similarly, destroymethod specifies a method that is called just before a bean is removed from the container.

## Initialization callbacks

The org.springframework.beans.factory.InitializingBean interface specifies a single method −

void afterPropertiesSet() throws Exception;

Thus, you can simply implement the above interface and initialization work can be done inside afterPropertiesSet() method as follows −

public class ExampleBean implements InitializingBean {

public void afterPropertiesSet() {

// do some initialization work

}

}

In the case of XML-based configuration metadata, you can use the **init-method** attribute to specify the name of the method that has a void no-argument signature. For example −

<bean id = "exampleBean" class = "examples.ExampleBean" init-method = "init"/>

Following is the class definition −

public class ExampleBean {

public void init() {

// do some initialization work

}

}

## Destruction callbacks

The *org.springframework.beans.factory.DisposableBean* interface specifies a single method −

void destroy() throws Exception;

Thus, you can simply implement the above interface and finalization work can be done inside destroy() method as follows −

public class ExampleBean implements DisposableBean {

public void destroy() {

// do some destruction work

}

}

In the case of XML-based configuration metadata, you can use the **destroy-method** attribute to specify the name of the method that has a void no-argument signature. For example −

<bean id = "exampleBean" class = "examples.ExampleBean" destroy-method = "destroy"/>

Following is the class definition −

public class ExampleBean {

public void destroy() {

// do some destruction work

}

}

If you are using Spring's IoC container in a non-web application environment; for example, in a rich client desktop environment, you register a shutdown hook with the JVM. Doing so ensures a graceful shutdown and calls the relevant destroy methods on your singleton beans so that all resources are released.

The **BeanPostProcessor** interface defines callback methods that you can implement to provide your own instantiation logic, dependency-resolution logic, etc. You can also implement some custom logic after the Spring container finishes instantiating, configuring, and initializing a bean by plugging in one or more BeanPostProcessor implementations.

You can configure multiple BeanPostProcessor interfaces and you can control the order in which these BeanPostProcessor interfaces execute by setting the **order** property provided the BeanPostProcessor implements the **Ordered**interface.

The BeanPostProcessors operate on bean (or object) instances, which means that the Spring IoC container instantiates a bean instance and then BeanPostProcessor interfaces do their work.

An **ApplicationContext** automatically detects any beans that are defined with the implementation of the **BeanPostProcessor** interface and registers these beans as postprocessors, to be then called appropriately by the container upon bean creation

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

BeanPostProcessor, which prints a bean name before and after initialization of any bean. You can implement more complex logic before and after instantiating a bean because you have access on bean object inside both the post processor methods.

mport org.springframework.beans.factory.config.BeanPostProcessor;

import org.springframework.beans.BeansException;

public class InitHelloWorld implements BeanPostProcessor {

public Object postProcessBeforeInitialization(Object bean, String beanName)

throws BeansException {

System.out.println("BeforeInitialization : " + beanName);

return bean; // you can return any other object as well

}

public Object postProcessAfterInitialization(Object bean, String beanName)

throws BeansException {

System.out.println("AfterInitialization : " + beanName);

return bean; // you can return any other object as well

}

}