**Spring IoC Containers**

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 lifecycle from creation till destruction. The Spring container uses dependency injection (DI) to manage the components that make up an application.

The container gets its instructions on what objects to instantiate, configure, and assemble by reading configuration metadata provided. The configuration metadata can be represented either by XML, Java annotations, or Java code. The following diagram is 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.



Spring provides following two distinct types of containers

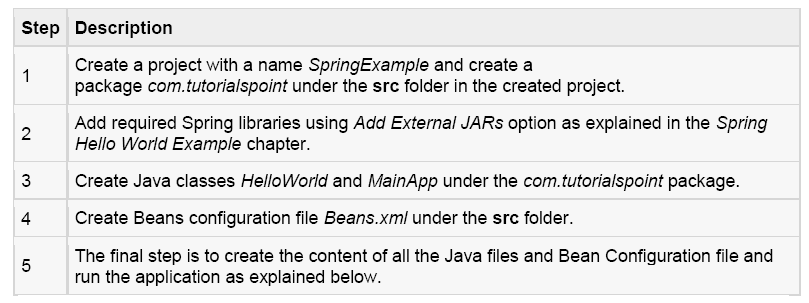
1. **Spring BeanFactory Container**

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.

There are a number of implementations of the BeanFactory interface that come supplied straight out-of-the-box with Spring. The most commonly used BeanFactory implementation is the **XmlBeanFactory**class. This container reads the configuration metadata from an XML file and uses it to create a fully configured system or application.

The BeanFactory is usually preferred where the resources are limited like mobile devices or applet based applications. So use an ApplicationContext unless you have a good reason for not doing so.

**Example :**Let us have working Eclipse IDE in place and follow the following steps to create a Spring application:





**2). Spring ApplicationContext Container**

The Application Context is spring's more advanced container. Similar to BeanFactory it can load bean definitions, wire beans together and dispense beans upon request. Additionally 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.

The ApplicationContext includes all functionality of the BeanFactory, it is generally recommended over the BeanFactory. BeanFactory can still be used for light weight applications like mobile devices or applet based applications.

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

We already have seen an example on ClassPathXmlApplicationContext container in Spring Hello World Example, and we will talk more about XmlWebApplicationContext in a separate chapter when we will discuss web based Spring applications. So let see one example on FileSystemXmlApplicationContext.

****

There are following two important points to note about the main program:

**1.** First step is to create factory object where we used framework APIFileSystemXmlApplicationContext to create the factory bean after loading the bean configuration file from the given path. The FileSystemXmlApplicationContext() API takes care of creating and initializing all the objects ie. beans mentioned in the XML bean configuration file.

**2.** Second step is used to get 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 actual object. Once you have object, you can use this object to call any class method.

**3.** Following is the content of the bean configuration file Beans.xml

****

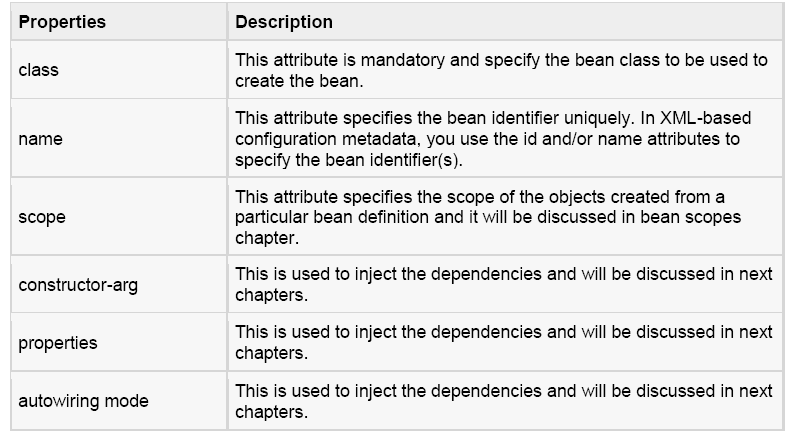
**Spring Bean Definition**

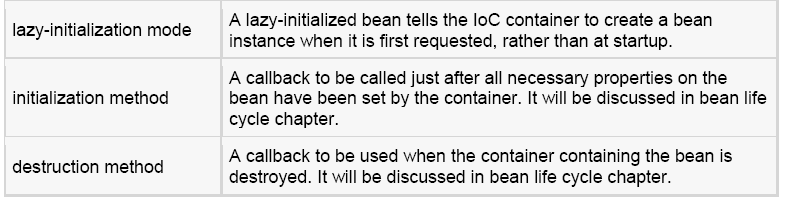
The objects that form the backbone of your application and that are managed by the Spring IoC container are called beans. A bean is an object that is instantiated, assembled, and otherwise managed by a Spring IoC container. These beans are created with the configuration metadata that you supply to the container, for example, in the form of XML <bean/> definitions which you have already seen in previous chapters.

The bean definition contains the information called **configuration metadata** which is needed for the container to know the followings:

* 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.

****

****

**Spring Configuration Metadata :**

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

1. XML based configuration file.

2. Annotation-based configuration

3. Java-based configuration

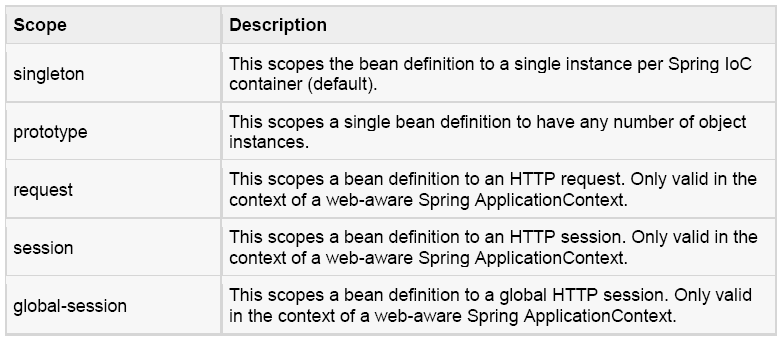
` Sample of XML based configuration file with different bean definitions including lazy initialization, initialization method and destruction method:

****

**Spring Bean Scopes**

When defining a <bean> in Spring, you have the option of declaring a scope for that bean. For example, To force Spring to produce a new bean instance each time one is needed, you should declare the bean's scope attribute to be **prototype**. Similar way if you want Spring to return the same bean instance each time one is needed, you should declare the bean's scope attribute to be **singleton**.

The Spring Framework supports following five scopes, three of which are available only if you use a web-aware ApplicationContext.

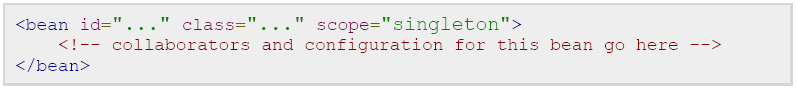
****

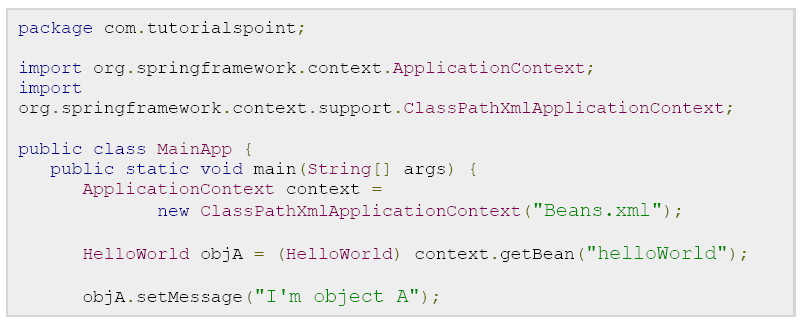
**The singleton scope:**

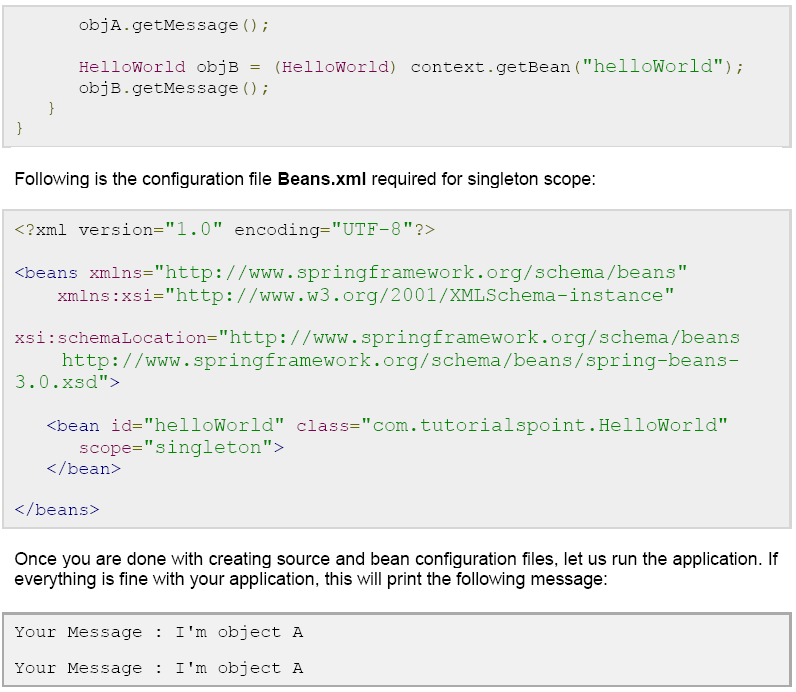
If scope is set to singleton, the Spring IoC container creates exactly one instance of the object defined by that bean definition. This single instance is stored in a cache of such singleton beans, and all subsequent requests and references for that named bean return the cached object.

The default scope is always singleton however, when you need one and only one instance of a bean, you can set the **scope** property to **singleton** in the bean configuration file, as shown below:

****

****

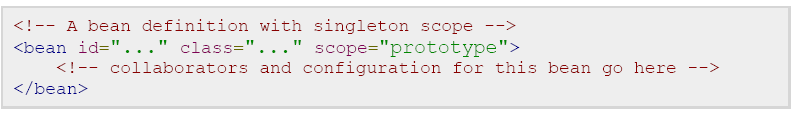
****

****

**The prototype scope :**

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.

To define a prototype scope, you can set the **scope** property to **prototype** in the bean configuration file, as shown below:

****

Once you are done with creating source and bean configuration files, let us run the application. If everything is fine with your application, this will print the following message:

****

**Spring Bean Life Cycle**

The life cycle of a Spring bean is easy to understand. 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 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 lifecycle callback methods which are required at the time of bean initialization and its destruction.

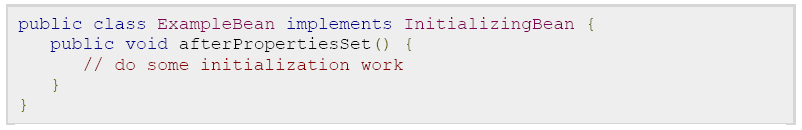
To define setup and teardown for a bean, we simply declare the <bean> with **init-method** and/or**destroy-method** parameters. The init-method attribute specifies a method that is to be called on the bean immediately upon instantiation. Similarly, destroy-method specifies a method that is called just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

**Initialization callbacks:**

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

****

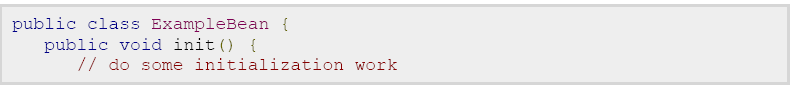
So you can simply implement above interface and initialization work can be done inside afterPropertiesSet() method as follows:

****

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:

****

Following is the class definition:

****

}

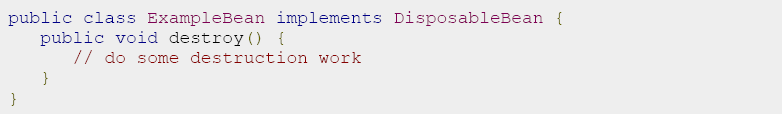
}

**Destruction callbacks:**

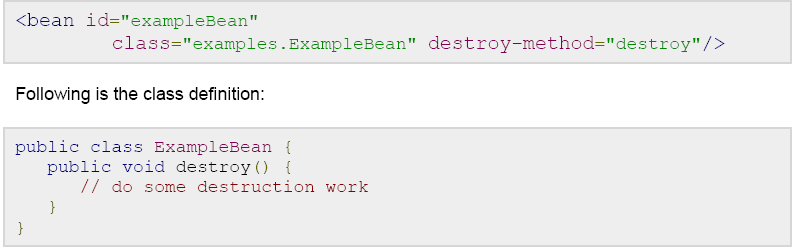
The *org.springframework.beans.factory.DisposableBean* interface specifies a single method:



So you can simply implement above interface and finalization work can be done inside destroy() method as follows:



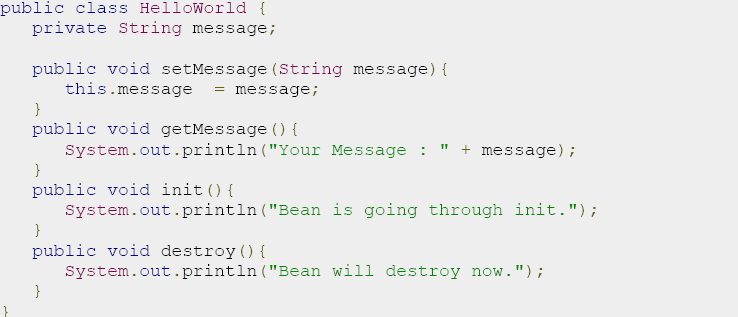
In the case of XML-based configuration metadata, you can use the **destroy-method** attribute to specify the name of the method that has a void no-argument signature. For example:



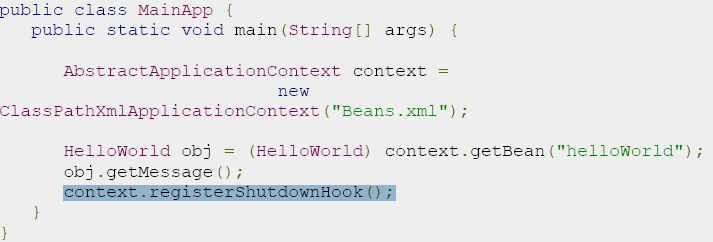
If you are using Spring's IoC container in a non-web application environment; for example, in a rich client desktop environment; you register a shutdown hook with the JVM. Doing so ensures a graceful shutdown and calls the relevant destroy methods on your singleton beans so that all resources are released.

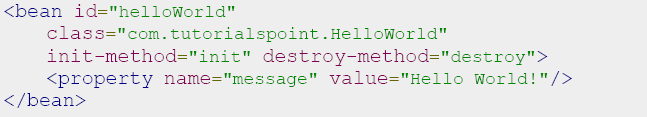
It is recommended that you do not use the InitializingBean or DisposableBean callbacks, because XML configuration gives much flexibility in terms of naming your method.

Example :

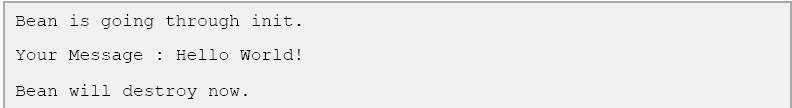


Following is the content of the **MainApp.java** file. Here you need to register a shutdown hook **registerShutdownHook()** method that is declared on the AbstractApplicationContext class. This will ensures a graceful shutdown and calls the relevant destroy methods.





**Output:**

****

**Default initialization and destroy methods :**

If you have too many beans having initialization and or destroy methods with the same name, you don't need to declare **init-method** and **destroy-method** on each individual bean. Instead framework provides the flexibility to configure such situation using **default-init-method** and **default-destroy-method**attributes on the <beans> element as follows:

o