# 1) What is Spring?

|  |
| --- |
| org.springframework |
|

It is a lightweight, [inversion of control](https://en.wikipedia.org/wiki/Inversion_of_control) [container](https://en.wikipedia.org/wiki/Servlet_container) ,loosely coupled and integrated framework for developing enterprise applications in java.

Spring framework was initially written by Rod Johnson and was first released under the Apache 2.0 license in June 2003.

Spring framework is said to be a non-invasive means it doesn’t force a programmer to extend or implement their class from any predefined class or interface given by Spring API, in struts we used to extend Action Class right that’s why struts is said to be invasive

In case of struts framework, it will forces the programmer that, the programmer class must extend from the base class provided by struts API

# 2) What are the advantages of spring framework?

1. Predefined Templates
2. Loose Coupling
3. Easy to test
4. Lightweight
5. Fast Development
6. Powerful Abstraction
7. Declarative support

#### 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.

# 3) What are the benefits of Spring Framework?

Following are the benefits of Spring framework:

* Extensive usage of Components
* Reusability
* Decoupling
* Reduces coding effort by using pattern implementations such as singleton, factory, service locator etc.
* Removal of leaking connections
* Declarative transaction management
* Easy to integrate with third party tools and technologies.

# 4)What are the differences between EJB and Spring ?

   Spring and EJB feature comparison.TransactionManagement, Declarative transaction support, Persistence, Declarative security, Distributed computing.

|  |  |  |
| --- | --- | --- |
| **Feature** | **EJB** | **Spring** |
| Transaction management | * Must use a JTA transaction manager. * Supports transactions that span remote method calls. | * Supports multiple transaction environments through its PlatformTransactionManager interface, including JTA, Hibernate, JDO, and JDBC. * Does not natively support distributed transactions—it must be used with a JTA transaction manager. |
| Declarative transaction support | * Can define transactions declaratively through the deployment descriptor. * Can define transaction behavior per method or per class by using the wildcard character \*. * Cannot declaratively define rollback behavior—this must be done programmatically. | * Can define transactions declaratively through the Spring configuration file or through class metadata. * Can define which methods to apply transaction behavior explicitly or by using regular expressions. * Can declaratively define rollback behavior per method and per exception type. |
| Persistence | Supports programmatic bean-managed persistence and declarative container managed persistence. | Provides a framework for integrating with several persistence technologies, including JDBC, Hibernate, JDO, and iBATIS. |
| Declarative security | * Supports declarative security through users and roles. The management and implementation of users and roles is container specific. * Declarative security is configured in the deployment descriptor. | * No security implementation out-of-the box. * Acegi, an open source security framework built on top of Spring, provides declarative security through the Spring configuration file or class metadata. |
| Distributed computing | Provides container-managed remote method calls. | Provides proxying for remote calls via RMI, JAX-RPC, and web services. |

Spring 4 features:

# 5) Here are the some important improvements in Spring 4.x

* Removed Deprecated Packages and Methods, [check out the API Differences Report](http://docs.spring.io/spring-framework/docs/3.2.4.RELEASE_to_4.0.0.RELEASE/)
* Java 8 Support[
* JSR-335 [Lambda expressions](https://javabeat.net/2012/05/a-sneak-peak-at-the-lambda-expressions-in-java-8/)
* JSR-310 Date-Time value types for Spring data binding and formatting.
* Java EE 6 and 7 or above is now considered the baseline for Spring Framework 4

Java EE 7 Enterprise Concurrency support.

* Groovy Bean Definition DSL, [read more about this API](http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/beans/factory/groovy/GroovyBeanDefinitionReader.html).
* Core Container Improvements
* General Web Improvements

WebSocket, SockJS, and STOMP Messaging[ STOMP (Simple (or Streaming) Text-Oriented Messaging Protocol) is a simple text-oriented protocol, similar to HTTP].

* [JSR-343](http://jcp.org/en/jsr/detail?id=343) JMS 2.0. latest version
* [JSR-338](http://jcp.org/en/jsr/detail?id=338) JPA 2.1. latest version
* Testing Improvements

**And Spring 3.x which introduced lot of new features like:**

* Spring MVC Test Framework
* Asynchronous MVC processing on Servlet 3.0
* custom @Bean definition annotations in @Configuration classes
* @Autowired and @Value to be used as meta-annotations
* Concurrency refinements across the framework
* loading WebApplicationContexts in the TestContext framework
* JCache 0.5 (JSR-107)

**1. Removed Deprecated Packages and Methods**

All deprecated packages, and many deprecated classes and methods have been removed with version 4.0. If you are upgrading from a previous release of Spring, you should ensure that you have fixed any deprecated calls that you were making to outdated APIs. This should be the challenging point for those who want to migrate their old spring applications to the new version. As a caution, if you are running the old spring application, then first thing to get rid of your deprecated APIs before migrating to the latest version.

For a complete set of changes, [check out the API Differences Report](http://docs.spring.io/spring-framework/docs/3.2.4.RELEASE_to_4.0.0.RELEASE/).

**2. Java 8 Support**

Spring Framework 4.0 provides support for several Java 8 features. You can make use of lambda expressions and method references with Spring’s callback interfaces. There is first-class support for java.time (JSR-310), and several existing annotations have been retrofitted as @Repeatable. You can also use Java 8’s parameter name discovery (based on the -parameters compiler flag) as an alternative to compiling your code with debug information enabled.

**Some of the Java SE 8 features to be supported include:**

* JSR-335 Lambda expressions.
* JSR-310 Date-Time value types for Spring data binding and formatting.
* Support for the new 1.8 byte code format (required to support Lambda expressions).

Spring 4.0 has increased the minimum recommendation to Java 6.0. Also it states that any new project recommended to use the Java 7.0 for their projects with Spring 4.0. It still supports the lower version Java 6 and 7 without any problem.

**3. Java EE 6 and 7**

Java EE version 6 or above is now considered the baseline for Spring Framework 4, with the JPA 2.0 and Servlet 3.0 specifications being of particular relevance. It is possible to run your application in Servlet 2.5, but it is recommended to use Servlet 3.0 environment.

**4. Groovy Bean Definition DSL**

With Spring Framework 4.0 it is now possible to define external bean configuration using a Groovy DSL. [Read more about this API](http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/beans/factory/groovy/GroovyBeanDefinitionReader.html).

**5. Core Container Improvements**

There have been several general improvements to the core container:

* Spring now treats [generic types as a form ofqualifier](http://docs.spring.io/spring/docs/4.0.0.RELEASE/spring-framework-reference/htmlsingle/#beans-generics-as-qualifiers) when injecting Beans. For example, if you are using a Spring Data Repository you can now easily inject a specific implementation: @Autowired Repository<Customer> customerRepository.
* If you use Spring’s meta-annotation support, you can now develop custom annotations that [expose specific attributes from the source annotation](http://docs.spring.io/spring/docs/4.0.0.RELEASE/spring-framework-reference/htmlsingle/#beans-meta-annotations).
* Beans can now be Ordered when they are [autowired intolists and arrays](http://docs.spring.io/spring/docs/4.0.0.RELEASE/spring-framework-reference/htmlsingle/" \l "beans-autowired-annotation" \t "_blank). Both the @Ordered annotation and Ordered interface are supported.
* The @Lazy annotation can now be used on injection points, as well as @Bean definitions.
* The [@Description annotation has been added](http://docs.spring.io/spring/docs/4.0.0.RELEASE/spring-framework-reference/htmlsingle/#beans-java-bean-description) for developers using Java-based configuration.
* A generalized model for [conditionally filtering beans](http://docs.spring.io/spring/docs/4.0.0.RELEASE/spring-framework-reference/htmlsingle/#beans-java-conditional) has been added via the @Conditional annotation. This is similar to @Profile but allows for user-defined strategies to be developed.
* [CGLIB-based proxy classes](http://docs.spring.io/spring/docs/4.0.0.RELEASE/spring-framework-reference/htmlsingle/#aop-pfb-proxy-types) no longer require a default constructor. Support is provided via the [objenesis](http://code.google.com/p/objenesis/" \t "_blank) library which is repackaged *inline* and distributed as part of the Spring Framework. With this strategy, no constructor at all is being invoked for proxy instances anymore.
* There is managed time zone support across the framework now, e.g. on LocaleContext.

**6. General Web Improvements**

Deployment to Servlet 2.5 servers remains an option, but Spring Framework 4.0 is now focused primarily on Servlet 3.0+ environments. If you are using the Spring MVC Test Framework you will need to ensure that a Servlet 3.0 compatible JAR is in your test classpath.

In addition to the WebSocket support mentioned later, the following general improvements have been made to Spring’s Web modules:

* You can use the new ***@RestController*** annotation with Spring MVC applications, removing the need to add ***@ResponseBody*** to each of your ***@RequestMapping*** methods.
* The ***AsyncRestTemplate***class has been added, allowing non-blocking asynchronous support when developing REST clients.
* Spring now offers comprehensive timezone support when developing Spring MVC applications.

**7. WebSocket, SockJS, and STOMP Messaging**

A new spring-websocket module provides comprehensive support for WebSocket-based, two-way communication between client and server in web applications. It is compatible with JSR-356, the Java WebSocket API, and in addition provides SockJS-based fallback options (i.e. WebSocket emulation) for use in browsers that don’t yet support the WebSocket protocol (e.g. Internet Explorer < 10).

A new spring-messaging module adds support for STOMP as the ***WebSocket***sub-protocol to use in applications along with an annotation programming model for routing and processing STOMP messages from ***WebSocket***clients. As a result an ***@Controller*** can now contain both ***@RequestMapping*** and ***@MessageMapping*** methods for handling HTTP requests and messages from WebSocket-connected clients. The new spring-messaging module also contains key abstractions from the Spring Integration project such as ***Message, MessageChannel, MessageHandler***, and others to serve as a foundation for messaging-based applications.

**8. Testing Improvements**

In addition to pruning of deprecated code within the spring-test module, Spring Framework 4.0 introduces several new features for use in unit and integration testing.

* Almost all annotations in the spring-test module (e.g., *@ContextConfiguration, @WebAppConfiguration, @ContextHierarchy, @ActiveProfiles, etc.*) can now be used as meta-annotations to create custom composed annotations and reduce configuration duplication across a test suite.
* Active bean definition profiles can now be resolved programmatically, simply by implementing a custom ***ActiveProfilesResolver***and registering it via the resolver attribute of *@ActiveProfiles*.
* A new ***SocketUtils***class has been introduced in the spring-core module which enables you to scan for free TCP and UDP server ports on localhost. This functionality is not specific to testing but can prove very useful when writing integration tests that require the use of sockets, for example tests that start an in-memory SMTP server, FTP server, Servlet container, etc.
* As of Spring 4.0, the set of mocks in the *org.springframework.mock.web* package is now based on the Servlet 3.0 API. Furthermore, several of the Servlet API mocks (e.g., **MockHttpServletRequest**, **MockServletContext**, etc.) have been updated with minor enhancements and improved configurability.

# 6) Spring versions List

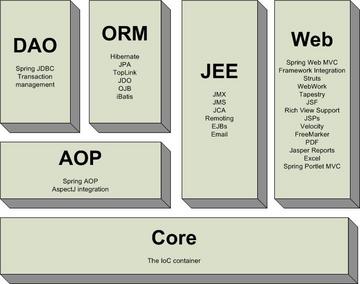
Minimum requirements

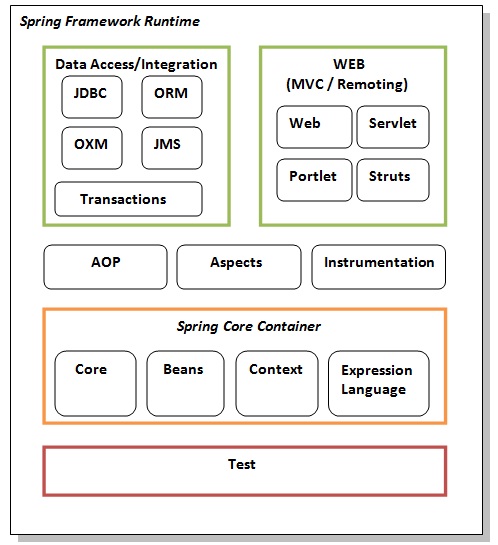
* JDK 8+ for Spring Framework 5.x
* JDK 6+ for Spring Framework 4.x
* JDK 5+ for Spring Framework 3.x

|  |  |  |
| --- | --- | --- |
| Unreleased | [5.1 RC1](https://jira.spring.io/browse/SPR/fixforversion/16630) | 08/May/18 |
| Unreleased | [5.0.5](https://jira.spring.io/browse/SPR/fixforversion/16634) | 03/Apr/18 |
| Unreleased | [5.0.4](https://jira.spring.io/browse/SPR/fixforversion/16629) | 26/Feb/18 |
| Released) | [5.0.3](https://jira.spring.io/browse/SPR/fixforversion/16496) | 23/Jan/18 |

# 7)Spring Modules, What Are Spring Modules

Actually in spring 1.x, the framework has divided into 7 well defined modules.  But in 2.x framework is divided into 6 modules only..





**Test**

This layer provides support of testing with JUnit and TestNG.

**Spring Core Container**

The Spring Core container contains core, beans, context and expression language (EL) modules.

**Core and Beans**

These modules provide IOC and Dependency Injection features.

**Context**

This module supports internationalization (I18N), EJB, JMS, Basic Remoting.

**Expression Language**

It is an extension to the EL defined in JSP. It provides support to setting and getting property values, method invocation, accessing collections and indexers, named variables, logical and arithmetic operators, retrieval of objects by name etc.

**AOP, Aspects and Instrumentation**

These modules support aspect oriented programming implementation where you can use Advices, Pointcuts etc. to decouple the code.

The aspects module provides support to integration with AspectJ.

The instrumentation module provides support to class instrumentation and classloader implementations.

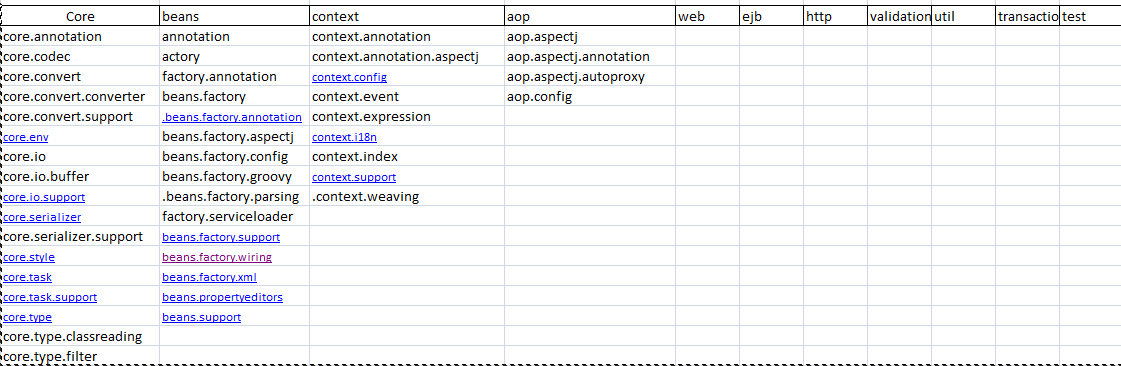
**Data Access / Integration**

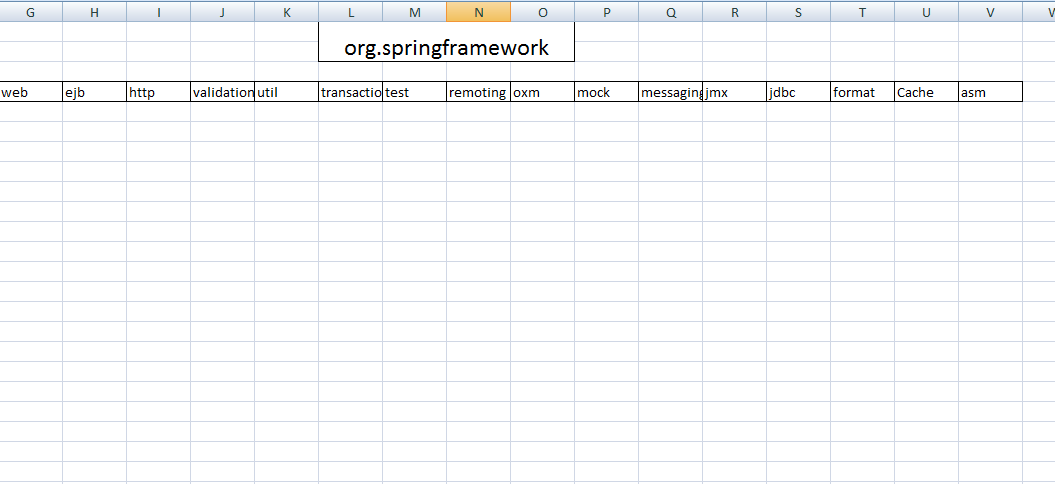
This group comprises of JDBC, ORM, OXM, JMS and Transaction modules. These modules basically provide support to interact with the database.

**Web**

This group comprises of Web, Web-Servlet, Web-Struts and Web-Portlet. These modules provide support to create web application.

# 8) spring package overview





# 9)List of Core/common GoF / J2EE Design Patterns which are internally used by Spring / based upon:

1. **MVC** - The advantage with Spring MVC is that your controllers are POJOs as opposed to being servlets. This makes for easier testing of controllers. One thing to note is that the controller is only required to return a logical view name, and the view selection is left to a separate ViewResolver. This makes it easier to reuse controllers for different view technologies.

2. **Front controller** - Spring provides "DispatcherServlet" to ensure an incoming request gets dispatched to your controllers.

3. **View Helper** - Spring has a number of custom JSP tags, and velocity macros, to assist in separating code from presentation in views.

4. **Singleton** - Beans defined in spring config files are singletons by default.

5. **Prototype** - Instance type can be prototype.

6. **Factory** - Used for loading beans through BeanFactory and Application context.

7. **Builder** - Spring provides programmatic means of constructing BeanDefinitions using the builder pattern through Class "BeanDefinitionBuilder".

8. **Template** - Used extensively to deal with boilerplate repeated code (such as closing connections cleanly, etc..). For example JdbcTemplate.

9. **Proxy** - Used in AOP & Remoting.

10. **DI/IOC** - It is central to the whole BeanFactory/ApplicationContext stuff.

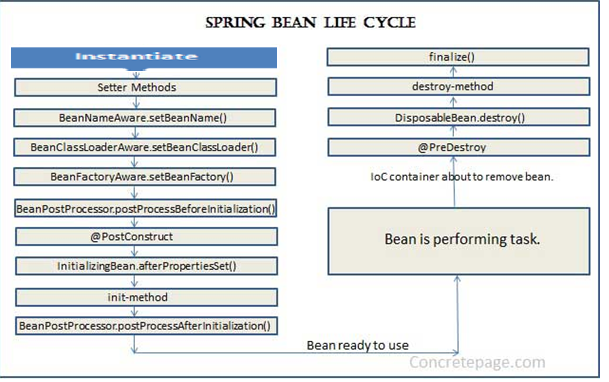
# 10)Overview of Spring Bean Life Cycle

Spring Beans are Instantiated / Managed by [**Spring IoC Container**](http://javabeginnerstutorial.com/spring-framework-tutorial/spring-ioc-container-overview/).  These beans can be created by providing bean specific *configuration metadata* to container. Configuration Metadata can be provided in any of below formats.

1. *XML*
2. *Annotation*
3. *Java Code*

Spring framework provides following **4 ways for controlling life cycle events** of bean:

1. **InitializingBean and DisposableBean callback interfaces**
2. **Other Bean Name, bean factory and Application Context  Aware interfaces for specific behavior**
3. **custom init() and destroy() methods in bean configuration file**
4. **@PostConstruct and @PreDestroy annotations**



Standard bean lifecycle interfaces & there standard order of execution are given below.

  1 IoC container will look for the configuration metadata of given Bean.  
  2 Once find, container will create the instance of Bean(Using reflection API).  
  3 After instance creation dependency will be injected(DI).

4 setBeanName method of **BeanNameAware I**nterface. It sets the name of the bean in the bean factory that created this bean.

5 setBeanFactory  method of **BeanFactoryAware**Interface. Callback that supplies the owning factory to a bean instance

6 setApplicationContext method of **ApplicationContextAware**Interface. Set the ApplicationContext that this object runs in.

7 IoC container calls BeanPostProcessor.postProcessBeforeInitialization on the bean. Using this method a wrapper can be applied on original bean..before any bean initialization callbacks.

8 Now the method annotated with @PostConstruct is called.

9 After @PostConstruct, the method InitializingBean.afterPropertiesSet() is called.

10 Now the method specified by init-method attribute of bean in XML configuration is called.

11 postProcessAfterInitialization methods of BeanPostProcessors. Apply this BeanPostProcessor to the given new bean instance after any bean initialization callbacks

12 Now the bean instance is ready to be used. Perform the task using the bean.

13 Now when the ApplicationContext shuts down such as by using registerShutdownHook() then the method annotated with @PreDestroy is called.

14 After that DisposableBean.destroy() method is called on the bean.

15 Now the method specified by destroy-method attribute of bean in XML configuration is called.

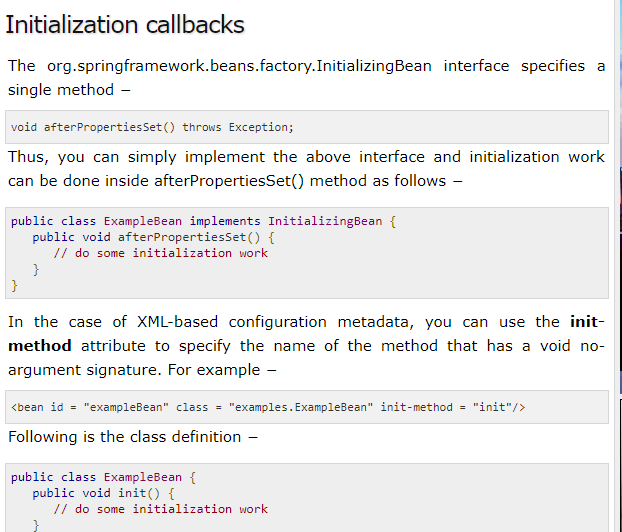
16 Before garbage collection, finalize() method of Object is called.

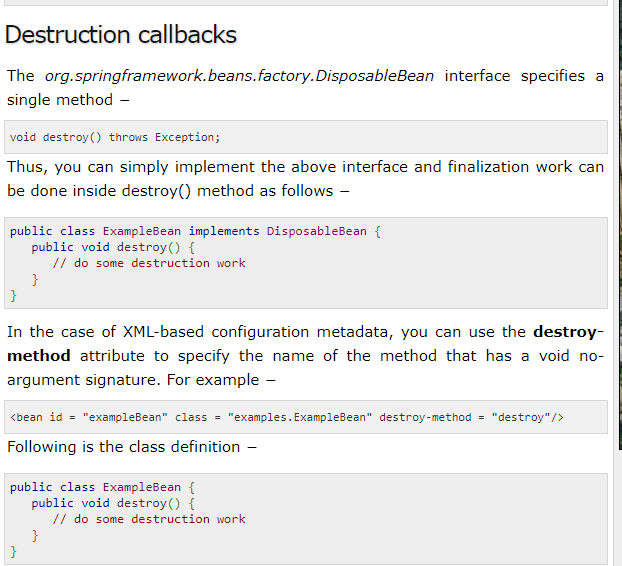
Detailed explanation

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.





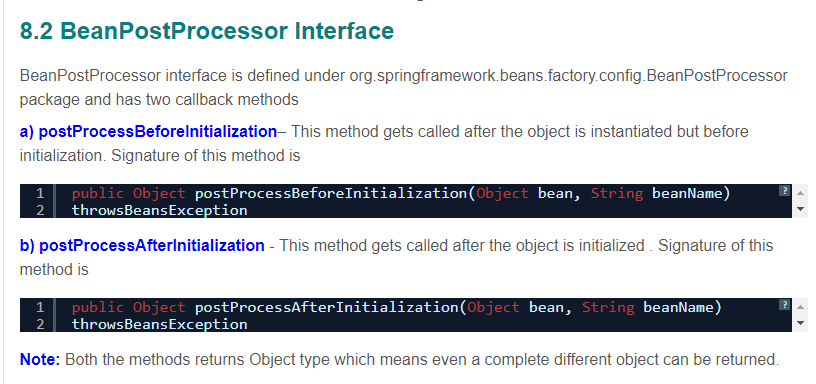
**Spring Bean Life Cycle – @PostConstruct, @PreDestroy Annotations**

Spring 2.5 onwards we can use annotations to specify life cycle methods using @PostConstruct and @PreDestroy annotations.

Spring framework also support @PostConstruct and @PreDestroy annotations for defining post-init and pre-destroy methods. These annotations are part of javax.annotation package. However for these annotations to work, we need to configure our spring application to look for annotations. We can do this either by defining bean of type org.springframework.context.annotation.CommonAnnotationBeanPostProcessor or by context:annotation-config element in spring bean configuration file.

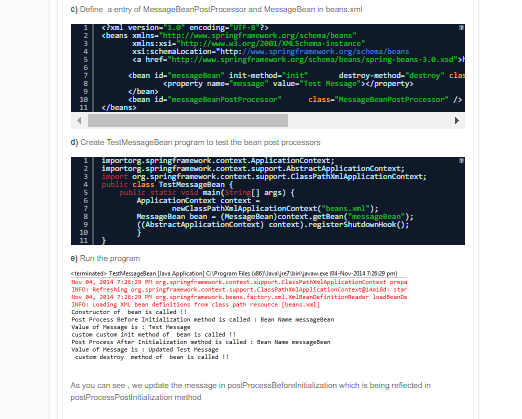


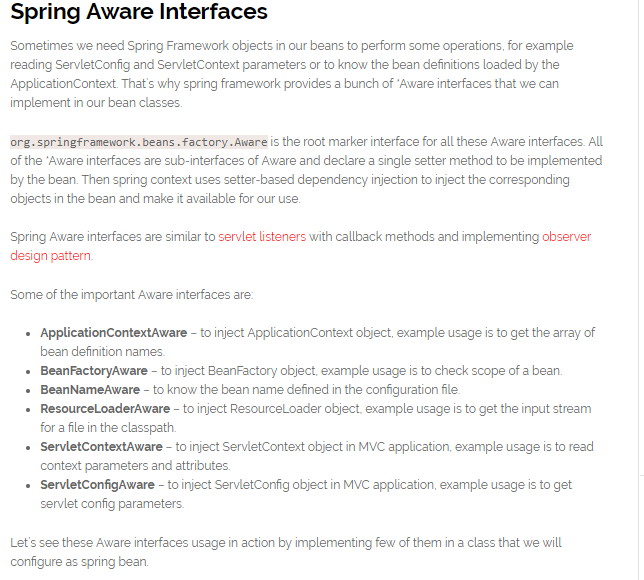




Example Code:









package com.journaldev.spring.service;

import java.util.Arrays;

import org.springframework.beans.BeansException;

import org.springframework.beans.factory.BeanClassLoaderAware;

import org.springframework.beans.factory.BeanFactory;

import org.springframework.beans.factory.BeanFactoryAware;

import org.springframework.beans.factory.BeanNameAware;

import org.springframework.context.ApplicationContext;

import org.springframework.context.ApplicationContextAware;

import org.springframework.context.ApplicationEventPublisher;

import org.springframework.context.ApplicationEventPublisherAware;

import org.springframework.context.EnvironmentAware;

import org.springframework.context.ResourceLoaderAware;

import org.springframework.context.annotation.ImportAware;

import org.springframework.core.env.Environment;

import org.springframework.core.io.Resource;

import org.springframework.core.io.ResourceLoader;

import org.springframework.core.type.AnnotationMetadata;

public class MyAwareService implements ApplicationContextAware,

ApplicationEventPublisherAware, BeanClassLoaderAware, BeanFactoryAware,

BeanNameAware, EnvironmentAware, ImportAware, ResourceLoaderAware {

@Override

public void setApplicationContext(ApplicationContext ctx)

throws BeansException {

System.out.println("setApplicationContext called");

System.out.println("setApplicationContext:: Bean Definition Names="

+ Arrays.toString(ctx.getBeanDefinitionNames()));

}

@Override

public void setBeanName(String beanName) {

System.out.println("setBeanName called");

System.out.println("setBeanName:: Bean Name defined in context="

+ beanName);

}

@Override

public void setBeanClassLoader(ClassLoader classLoader) {

System.out.println("setBeanClassLoader called");

System.out.println("setBeanClassLoader:: ClassLoader Name="

+ classLoader.getClass().getName());

}

@Override

public void setResourceLoader(ResourceLoader resourceLoader) {

System.out.println("setResourceLoader called");

Resource resource = resourceLoader.getResource("classpath:spring.xml");

System.out.println("setResourceLoader:: Resource File Name="

+ resource.getFilename());

}

@Override

public void setImportMetadata(AnnotationMetadata annotationMetadata) {

System.out.println("setImportMetadata called");

}

@Override

public void setEnvironment(Environment env) {

System.out.println("setEnvironment called");

}

@Override

public void setBeanFactory(BeanFactory beanFactory) throws BeansException {

System.out.println("setBeanFactory called");

System.out.println("setBeanFactory:: employee bean singleton="

+ beanFactory.isSingleton("employee"));

}

@Override

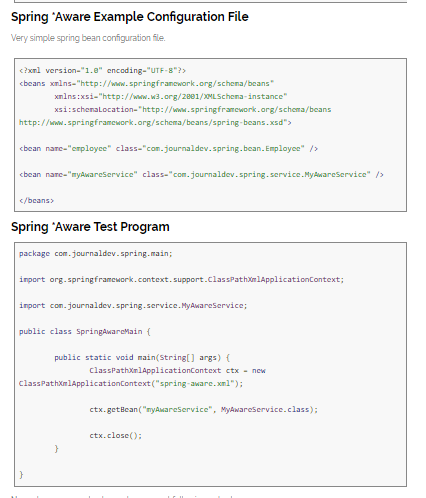
public void setApplicationEventPublisher(

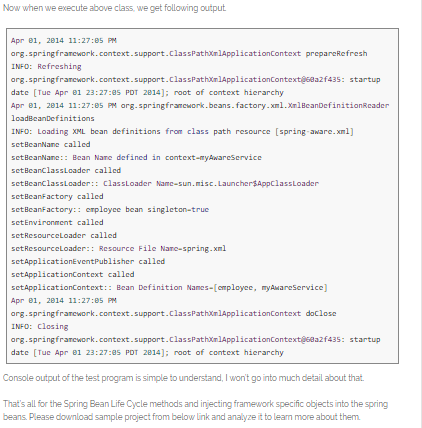
ApplicationEventPublisher applicationEventPublisher) {

System.out.println("setApplicationEventPublisher called");

}

}





# 11) Sprin IoC Container

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:

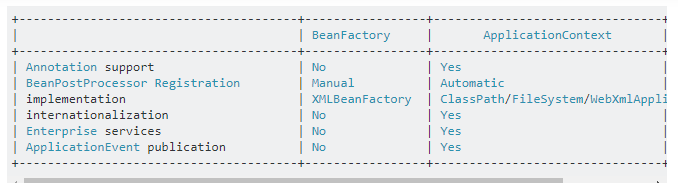
1. **BeanFactory [**org.springframework.beans.factory.BeanFactory]

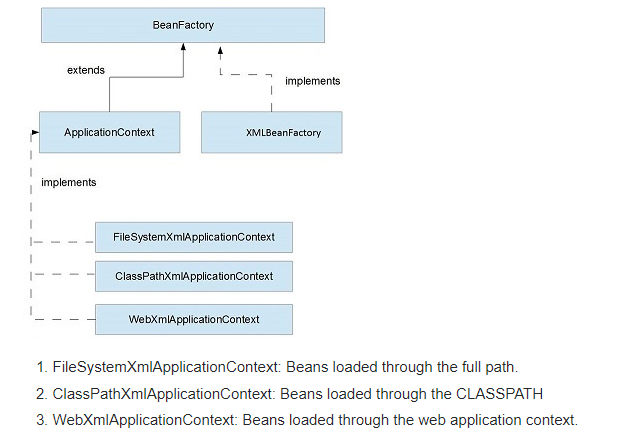
Used for [Standalone applications]

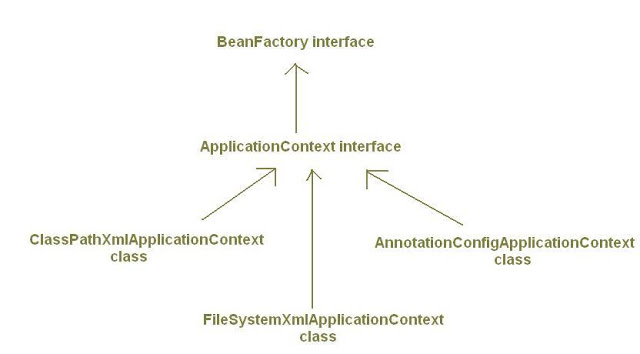
1. **ApplicationContext[**org.springframework.context.ApplicationContext]

## Differences between BeanFactory and ApplicationContext

|  |  |
| --- | --- |
| **BeanFactory** | **ApplicationContext** |
| BeanFactory: (Lazy container because it creates the objects for the beans only when you explicitly call from the user/main class) | ApplicationContext: (Eager container because of creating the objects of all singleton beans while loading the spring.xml file itself) |
| Does not support the Annotation based dependency Injection. | Support Annotation based dependency Injection.-@Autowired, @PreDestroy |
| Doesn't Support I18N | Support I18N |
| it doesn't allow configure to multiple configuration files. ex: BeanFactory context=new XmlBeanFactory(new Resource("applicationContext.xml")); | it allow to configure multiple configuration files. ex: ApplicationContext context=new ClasspathXmlApplicationContext("applicationContext.xml"); |
| BeanFactory Container is basic container, it can only create objects and inject Dependencies But we can’t attach other services like security, transaction, messaging etc | to provide all the services we have to use ApplicationContext Container. |
| Beanfactory Container support only two scopes (singleton & prototype) of the beans | But ApplicationContext Container supports all the beans scope. |
|  |  |







## SImplementations of ApplicationContext:

* There are many implementations of *ApplicationContext* interface. Important ones are:

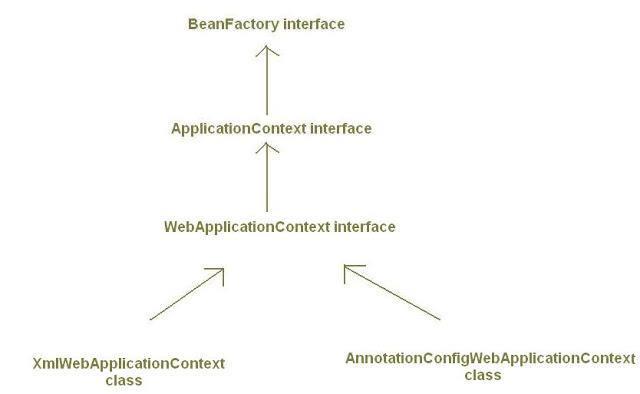
## 1)  ClassPathXmlApplicationContext:

* It loads bean definitions from XML files located in the classpath.
* *Example1:*
* ApplicationContext context = new ClassPathXmlApplicationContext("myconfig.xml");
* *Example2:*  Loading configuration from multiple files under classpath.
* ApplicationContext context = new ClassPathXmlApplicationContext(
* newString[]{"servicesconfig.xml","daoconfig.xml"});

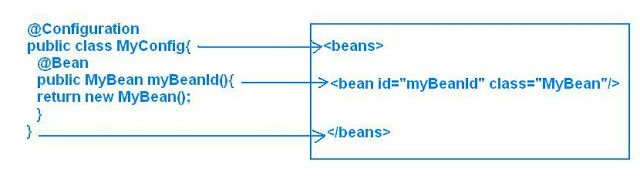
## 2) FileSystemXmlApplicationContext:

* It loads bean definitions from XML files in the file system.
* *Example:*
* ApplicationContext context = new FileSystemXmlApplicationContext("c:/myconfig.xml");

## 3)XmlWebApplicationContext:

* *XmlWebApplicationContext* is used to represent Spring Container for web applications.By defalut Spring creates object of *XmlWebApplicationContext* class to represent application context/spring container for web applications.
* It loads bean definitions from an XML file contained within a web application. By default it loads the configuration from file *"/WEB-INF/applicationContext.xml".*
* If we want to load bean definitions from more than one xml files we can specify their locations in  *contextConfigLocation*  parameter of *ContextLoaderListener* or *DispatcherServlet* in web.xml.[Read more about this here.](http://myjavadiary.blogspot.in/2012/08/webappicationcontext-and_5211.html)
* *XmlWebApplicationContext* is an implementation of *WebApplicationContext* interface which in turn extends *ApplicationContext*interface.
* [](http://1.bp.blogspot.com/--Ejm4KKBCk8/UC-Sr9zanoI/AAAAAAAAAGY/57DP2Z13L1A/s1600/applicationcontext_8.JPG)

## 4)AnnotationConfigApplicationContext:

* *AnnotationConfigApplicationContext* class is used when we are using *Java-based configuration* for the bean definitions*instead of Xml files*.
* In above *ApplicationContext* implementations (*ClassPathXmlApplicationContext, FileSystemXmlApplicationContext*) we have supplied bean configuration from xml configuration files. *AnnotationConfigApplicationContext* class is used to create Spring container which takes bean definitions from java classes annotated with @Configuration, instead of xml files.
* It is introduced in Spring 3.0.
* *Example*:
* public static void main(String[]args){
* /\* Creating Spring IoC Container Without XML configuration file\*/
* ApplicationContext context= new AnnotationConfigApplicationContext(MyConfig.class);  
    
  MyBean beanObj = context.getBean(MyBean.class);  
    
  beanObj.someMethod();
* }
* In above code, *AnnotationConfigApplicationContext* is accepting*MyConfig* class as input. Here we are obtaining bean definitions from a java class named *MyConfig* annotated with @Configuration, instead of a Xml file. *MyConfig* class is described as:
* @Configuration
* public class MyConfig{
* @Bean
* public MyBean myBeanId(){
* return new MyBean();
* }
* }
* By giving @Configuration annotation we are treating *Myconfig* class as  <beans></beans>  tag of xml file.
* By giving @Bean annotation we are treating myBean() method as  <bean id="..." class="..."/>
* [](http://1.bp.blogspot.com/-_80gXx0KxOo/UC_BXzbWZ0I/AAAAAAAAAGw/ZYOgrNCDYg8/s1600/applicationcontext_9_1.JPG)
* Name of myBeanId() method will be treated as *bean id*.
* Both @Configuration and @Bean are also introduced in Spring 3.0.

## *5)*AnnotationConfigWebApplicationContext*:*

* Like *XmlWebApplicationContext* is web counterpart for the *ClassPathXmlApplicationContext* and *FileSystemXmlApplicationContext* and is used to create application context for web  applications, similarly, *AnnotationConfigWebApplicationContext* is web counterpart for *AnnotationConfigApplicationContext.*
* *AnnotationConfigWebApplicationContext* is used to create application context for web applications by using java clases as input for bean definitions instead of xml files.
* By default Spring use *XmlWebApplicationContext* (an implementation of *WebApplicationContext*) for creating spring container in web applications. But we can change this default value to *AnnotationConfigWebApplicationContext* by changing the value of *contextClass* parameter of *ContextLoaderListener* or *DispatcherServlet* in web.xml as shown below:
* *For ContextLoaderListener:*
* <web-app>
* <context-param>
* <param-name>**contextClass**</param-name>
* <param-value>
* org.springframework.web.context.support.**AnnotationConfigWebApplicationContext**
* </param-value>
* </context-param>
* <context-param>
* <param-name>contextConfigLocation</param-name>
* <!--MyConfig must be annotated with @Configuration-->
* <param-value> MyConfig</param-value>
* </context-param>
* <listener>
* <listener-class> org.springframework.web.context.ContextLoaderListener </listener-class>
* </listener>
* </web-app>
* *For* *DispatcherServlet:*
* <web-app>
* <servlet>
* <servlet-name>mydispatcher</servlet-name>
* <servlet-class > org.springframework.web.servlet.DispatcherServlet
* </servlet-class>
* <init-param>
* <param-name>**contextClass**</param-name>
* <param-value>
* org.springframework.web.context.support.**AnnotationConfigWebApplicationContext**
* </param-value>
* </init-param>
* <init-param>
* <param-name>contextConfigLocation</param-name>
* <!--MyConfig must be class annotated with @Configuration-->
* <param-value> MyConfig </param-value>
* </init-param>
* </servlet>
* <servlet-mapping>
* <servlet-name>mydispatcher</servlet-name>
* <url-pattern>\*.htm</url-pattern>
* </servlet-mapping>
* </web-app>
* I would like to know your comments and if you liked the article then please share it on social networking buttons.

By default Spring use *XmlWebApplicationContext* (an implementation of *WebApplicationContext*) for creating spring container in web applications.

javax.servlet.ServletContext servletContext = ..

org.springframework.web.context.WebApplicationContext webApplicationContext = WebApplicationContextUtils.getWebApplicationContext(servletContext);

SampleService service = (SampleService) webApplicationContext.getBean("sampleService");

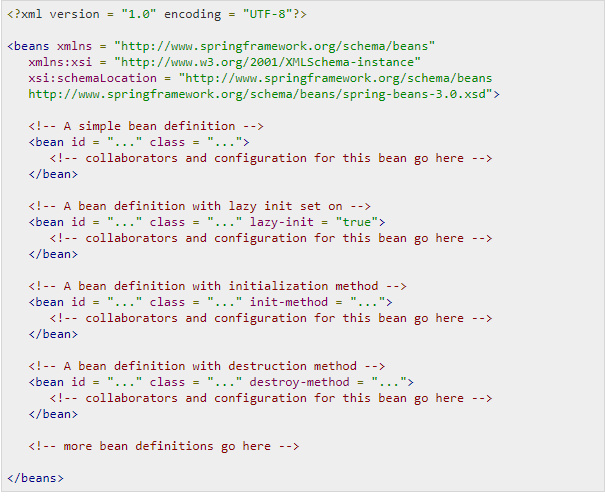
## Spring - Bean Definition

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

### XML based configuration file.

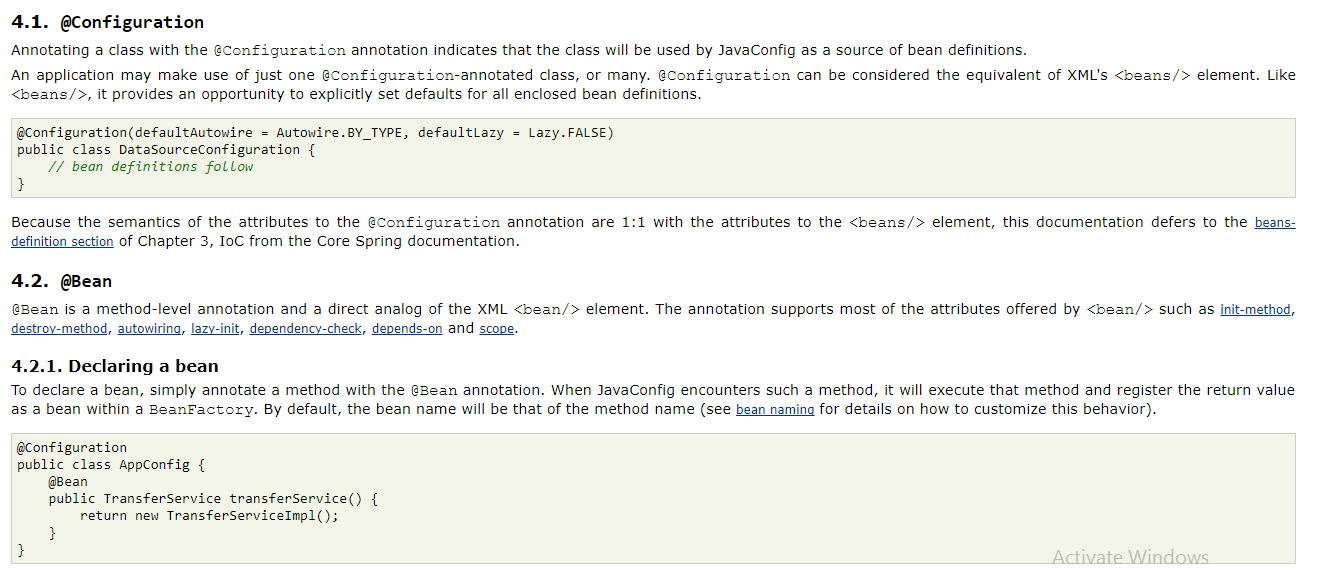
XML-based configuration file with different bean definitions including lazy initialization, initialization method, and destruction method

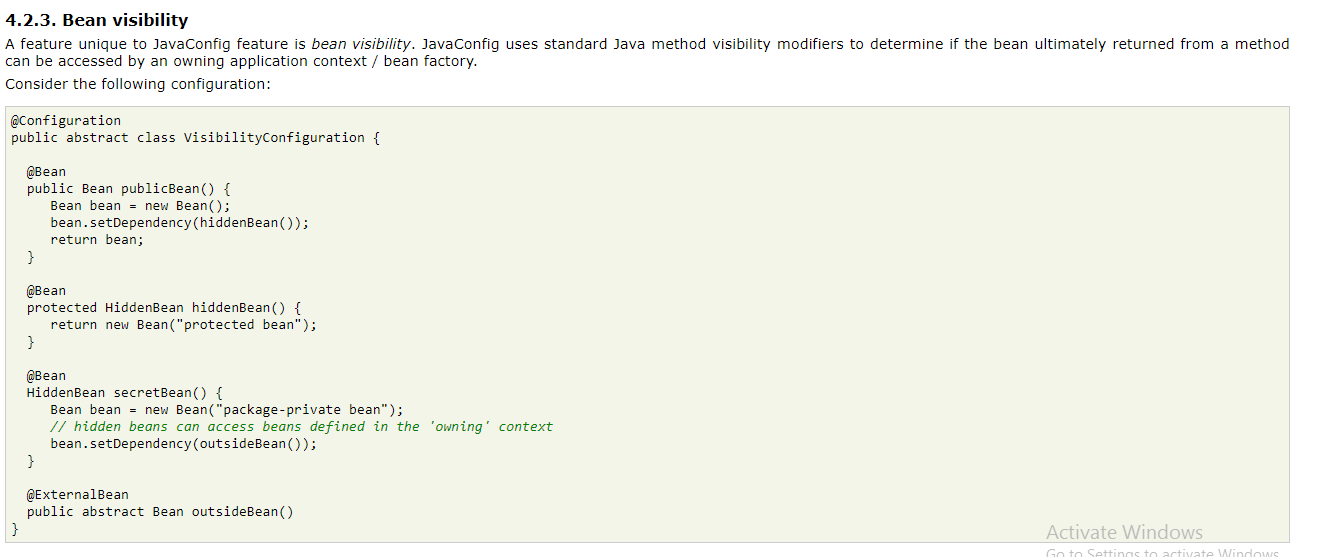


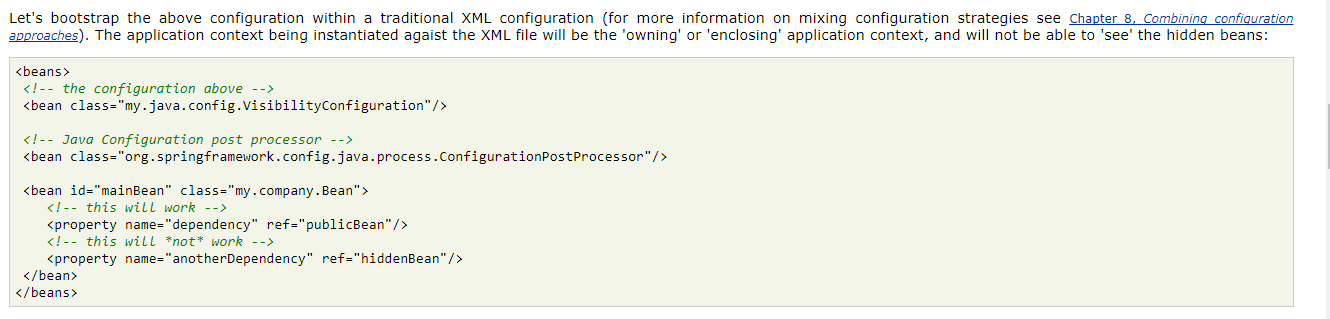
**XML Namespace(xmlns) is** a mechanism to avoid name conflicts by differentiating elements or attributes within an XML document that may have identical names, but different definitions.

**Xml:xsi:** Once you have the XML Schema Instance namespace available:**xmlns**:**xsi**="http://www.w3.org/2001/XMLSchema-instance" you can use the schemaLocation attribute. This attribute has two values, separated by a space. The first value is the namespace to use.

**xsi:schemaLocation** The xsi:schemaLocation attribute locates schemas for elements and attributes that are in a specified namespace.







|  |  |
| --- | --- |
| **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. |

# 12)Spring application has two types of context configuration files for [Spring MVC](https://javabeat.net/spring-mvc-example/) module:

1. **ApplicationContext (default name for this file is applicationContext.xml)**
2. **WebApplicationContext (default name for this file is xxx-servlet.xml where xxx is the DispatcherServlet name in web.xml)**

## ApplicationContext

* applicationContext.xml is the root context configuration for every web application.
* Spring loads applicationContext.xml file and creates the ApplicationContext for the whole application.
* There will be only one application context per web application.
* If you are not explicitly declaring the context configuration file name in web.xml using the contextConfigLocation param, Spring will search for the applicationContext.xml under WEB-INF folder and throw [FileNotFoundException](https://javabeat.net/filenotfoundexception-java/) if it could not find this file.

## WebApplicationContext

* Apart from **ApplicationContext**, there can be multiple **WebApplicationContext** in a single web application.
* In simple words, each DispatcherServlet associated with single WebApplicationContext.
* xxx-servlet.xml file is specific to the [DispatcherServlet](https://javabeat.net/spring-mvc-dispatcherservlet-example/) and a web application can have more than one DispatcherServlet configured to handle the requests.
* In such scenarios, each DispatcherServlet would have a separate xxx-servlet.xml configured. But, applicationContext.xml will be common for all the [servlet](https://javabeat.net/servlets-tutorials/)configuration files.
* Spring will by default load file named “xxx-servlet.xml” from your webapps WEB-INF folder where xxx is the servlet name in web.xml.
* If you want to change the name of that file name or change the location, add initi-param with contextConfigLocation as param name.

## ContextLoaderListener

* Performs the actual initialization work for the root application context.
* Reads a “contextConfigLocation” context-param and passes its value to the context instance, parsing it into potentially multiple file paths which can be separated by any number of commas and spaces, e.g. “WEB-INF/applicationContext1.xml, WEB-INF/applicationContext2.xml”.
* ContextLoaderListener is optional. Just to make a point here: you can boot up a Spring application without ever configuring ContextLoaderListener, just a basic minimum web.xml with DispatcherServlet.

Sample web.xml file with configurations:

<?xml version="1.0" encoding="UTF-8"?>

<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.com/xml/ns/javaee" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_3\_0.xsd" id="WebApp\_ID" version="3.0">

<display-name>Spring MVC</display-name>

<!-- This is the root application context for whole web application. -->

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/rootApplicationContext.xml</param-value>

</context-param>

<listener>

<listener-class>

org.springframework.web.context.ContextLoaderListener

</listener-class>

</listener>

<servlet>

<servlet-name>webmvc1</servlet-name>

<servlet-class>

org.springframework.web.servlet.DispatcherServlet

</servlet-class>

<!-- We require this configuration when we want to change the default name / location of the servlet specific configuration files -->

<init-param>

<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/mvc1-servlet.xml</param-value>

</init-param>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet>

<servlet-name>webmvc2</servlet-name>

<servlet-class>

org.springframework.web.servlet.DispatcherServlet

</servlet-class>

<!-- We require this configuration when we want to change the default name / location of the servlet specific configuration files -->

<init-param>

<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/mvc2-servlet.xml</param-value>

</init-param>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>webmvc1</servlet-name>

<url-pattern>/webmvc1</url-pattern>

</servlet-mapping>

<servlet-mapping>

<servlet-name>webmvc2</servlet-name>

<url-pattern>/webmvc2</url-pattern>

</servlet-mapping>

</web-app>

# 13)Spring - Bean Scopes

In the spring bean configurations, bean attribute called 'scope' defines what kind of object has to created and returned. There are 5 types of bean scopes available, they are:

**1) singleton:**Returns a single bean instance per Spring IoC container.

**2) prototype:**Returns a new bean instance each time when requested.

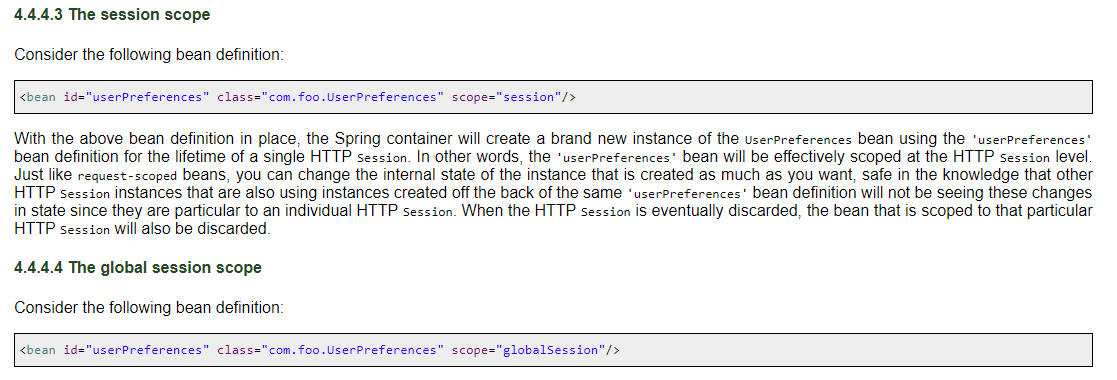
**3) request:**Returns a single instance for every HTTP request call.

**4) session:**Returns a single instance for every HTTP session.

**5) global session:**global session scope is equal as session scope on portlet-based web applications.

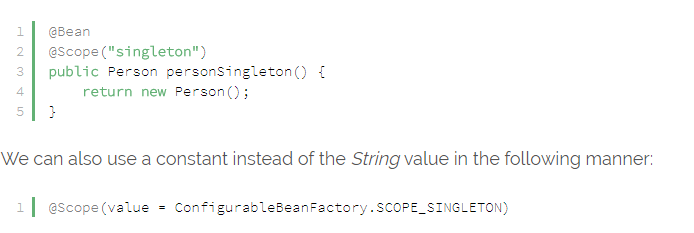
Here is an example for specifying bean scope configuration:

|  |  |  |
| --- | --- | --- |
| [?](http://www.java2novice.com/spring/bean-scope-types/)   |  |  | | --- | --- | | 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | <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">        <!-- Since scope is not defined, it assigns default value 'singleton'.       It creates only one instance per spring IoC. -->      <bean id="articleBean" class="com.java2novice.bean.Article" />        <!-- Since scope is prototype, it creates and returns ColorBean for every      call-->      <bean id="clrBean" class="com.java2novice.bean.ColorBean" scope="prototype"/>        <bean id="ticketBean" class="com.java2novice.bean.TicketManager" scope="singleton"/>    </beans> | |

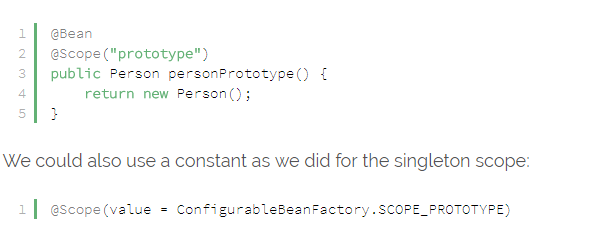


scope by using the **@Scope** annotation:

singleton



**Prototype Scope**



**Request Scope**

@Bean

@Scope(value = WebApplicationContext.SCOPE\_REQUEST, proxyMode = ScopedProxyMode.TARGET\_CLASS)

public HelloMessageGenerator requestMessage() {

    return new HelloMessageGenerator();

}

**Session Scope**

@Bean

@Scope(value = WebApplicationContext.SCOPE\_SESSION, proxyMode = ScopedProxyMode.TARGET\_CLASS)

public HelloMessageGenerator sessionMessage() {

    return new HelloMessageGenerator();

}

**GlobalSession Scope**

@Bean

@Scope(value = WebApplicationContext.SCOPE\_GLOBAL\_SESSION, proxyMode = ScopedProxyMode.TARGET\_CLASS)

public HelloMessageGenerator globalSessionMessage() {

    return new HelloMessageGenerator();

}

# ****14) Spring DI –Dependency Injection****

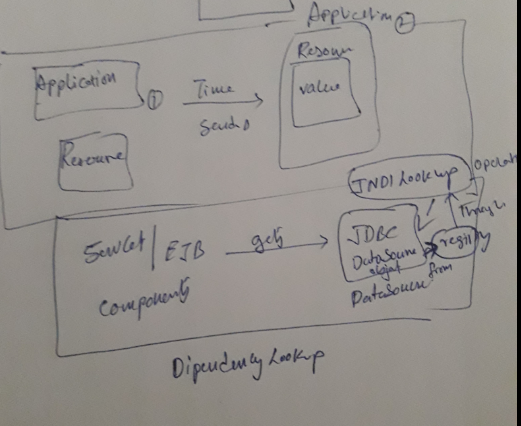
Dependency Injection (DI) design pattern is used to define the object dependencies between each other.

* Setter injection
* Construction injection
* Interface injection

But in spring we have only setter, constructor injections but **not** interface injection,  In spring 2.x we have interface injection

## WHAT IS THE Dependency lookup:

* + If resource of the application spending time to search and get dependent values from other resources of application  then it is called dependency lookup
  + **Example :**
    - The way servlet / ejb component gets jdbc data source object form registry through jndi lookup operation is called dependency lookup
  + In dependency lookup resource perform **“pull”** operation on other resource or on underlying s/w to get the dependent values
  + **Disadvantage**: resource has to spend time to search and get dependent values
  + **Advantage**: resource can get only required dependent values

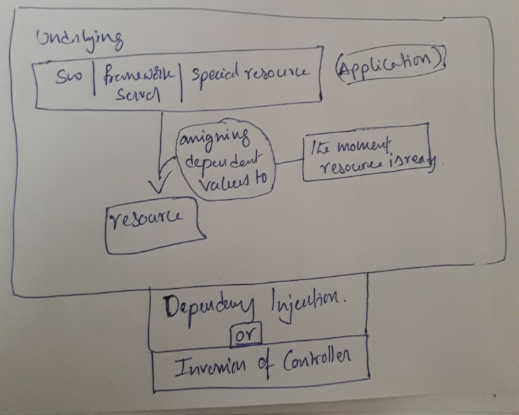


**Note**: in struts application form bean class is mediator for actionservlet to send multiple values of form pages to actionservlet class in the form of single object

## Dependency injection:

* + if underlying s/w (or) framework server (or) special resource of application assign dependent values to resource, the moment resource is ready then it is called dependency injection or inversion of controller (IOC)
  + **Example :**
    - The way student gets his course material the moment to register for course is called dependency injection
    - If jdbc data source object is assigned to servlet object by servlet container the moment servlet object is created is called dependency injection
  + In dependency injection underlying s/w or container/ framework server or special resource perform **“push”** operation on the resource of application to assign dependent values the moment resource is ready
  + **Advantage**: resource need not spend time to get dependent value and it can use dependent value at the moment resource is ready
  + **Disadvantage**: both necessary and unnecessary values will be injected
  + In struts the way action servlet dynamically assign form data to form bean class properties
  + By creating /locating formbean class object is called dependency injection or ioc

**Note**: the servlet container of web server / application server where struts application is deployed wills created actionservlet class object. This actionservlet create object for form bean and action class of struts application. The common thing the struts application contain is move architecture mvc design pattern



## Dependency Injection by Constructor Example

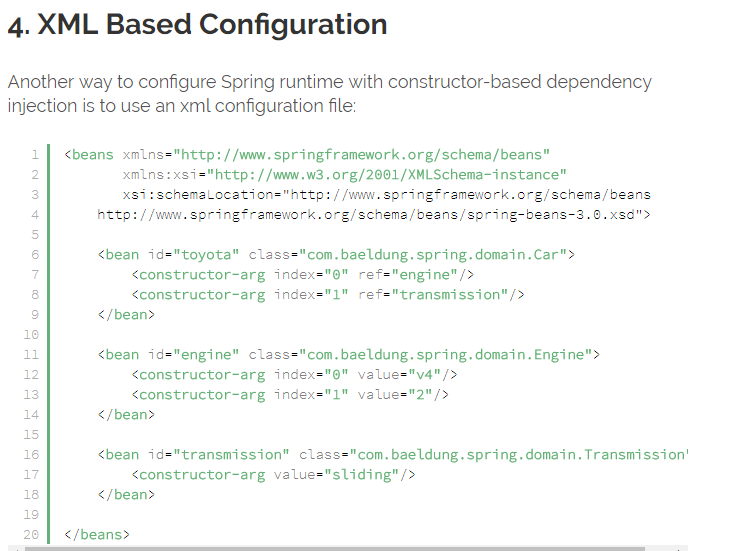
In this type of injection spring container uses constructor of the bean class for assigning the dependencies.

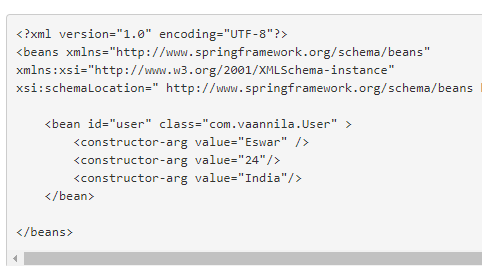
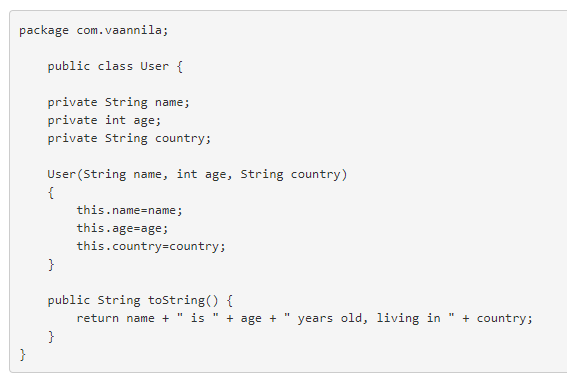
In spring config xml, we need to inform to the spring IOC container about constructor injection by using <constructor -arg />

The **<constructor-arg>**subelement of **<bean>** is used for constructor injection. Here we are going to inject

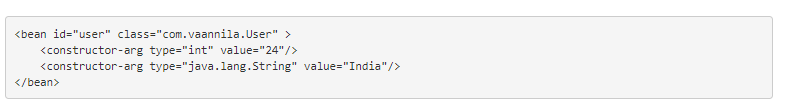
1. primitive and String-based values
2. Dependent object (contained object)
3. Collection values etc.

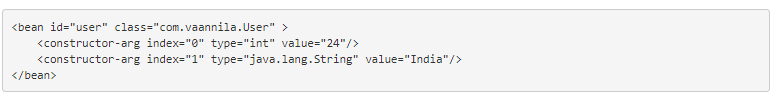
In spring bean class, if both constructor and setter injection applied for same property then constructor injection value will be overridden by setter injection, because constructor injection will happen at the object creation time, and setter after objection right…, so setter will overrides







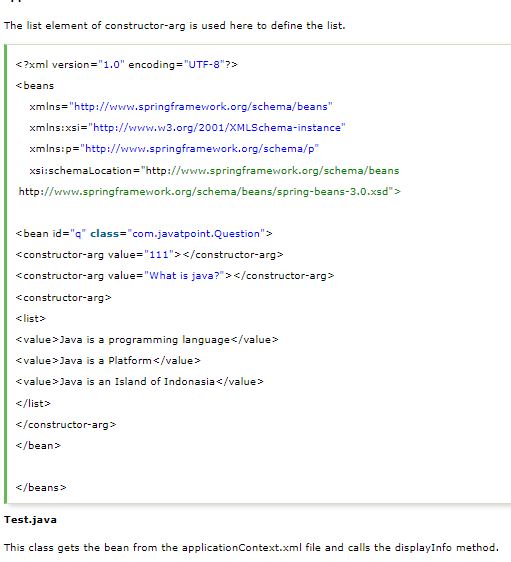
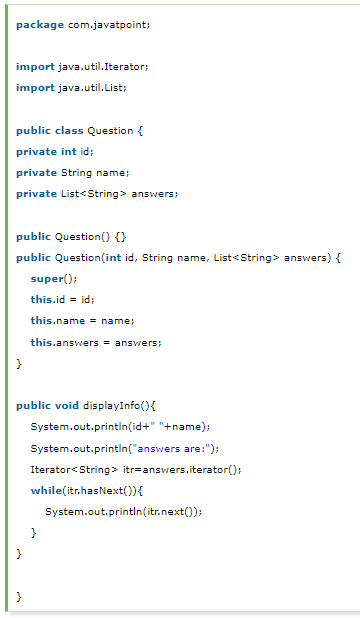




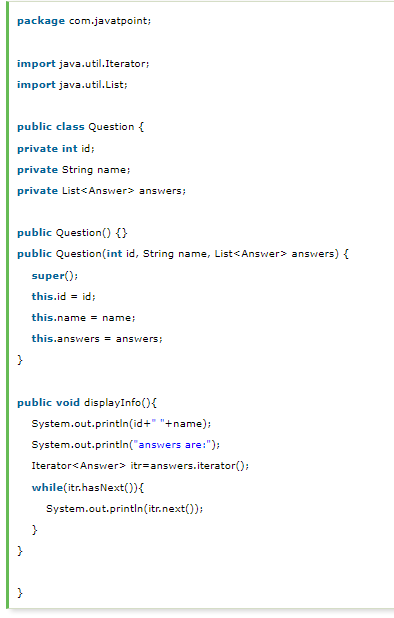
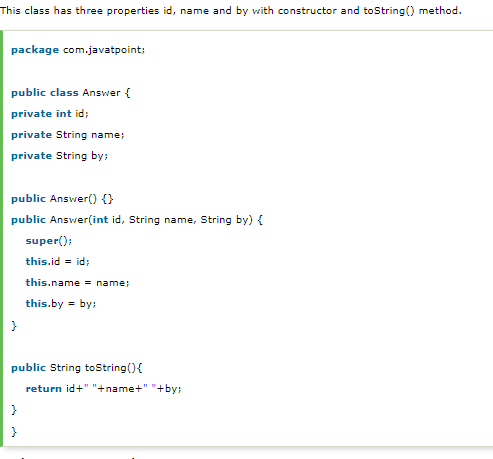
Collections definitions: List,Set,Map

## List or Set

### Constructor Injection with String Collection Example



### Constructor Injection with Non-String Collection (having Dependent Object) Example



## Map

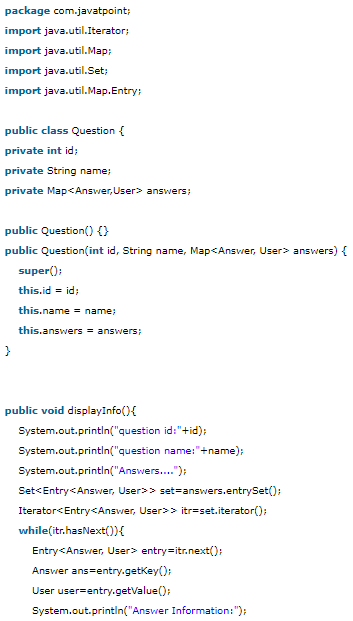
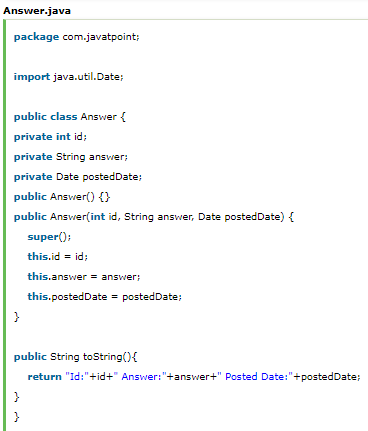
### Constructor Injection with Map Example

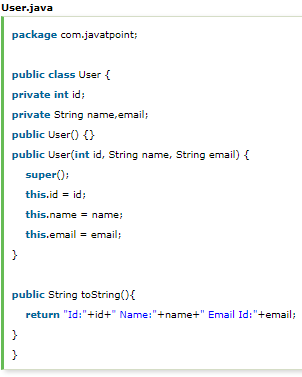
In this example, we are using **map** as the answer that have answer with posted username. Here, we are using key and value pair both as a string.

Like previous examples, it is the example of forum where **one question can have multiple answers**.

### Constructor Injection with Non-String Map (having dependent Object) Example

## Inheriting Bean in Spring

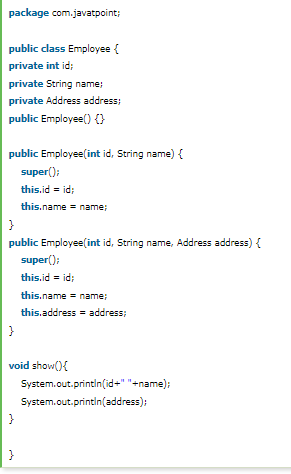
1. [Inheriting Bean in Spring](https://www.javatpoint.com/spring-tutorial-inheriting-bean-in-spring)

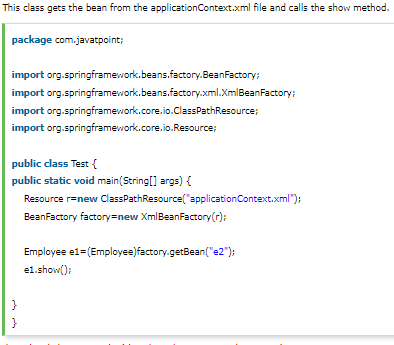
By using the **parent** attribute of **bean**, we can specify the inheritance relation between the beans. In such case, parent bean values will be inherited to the current bean.

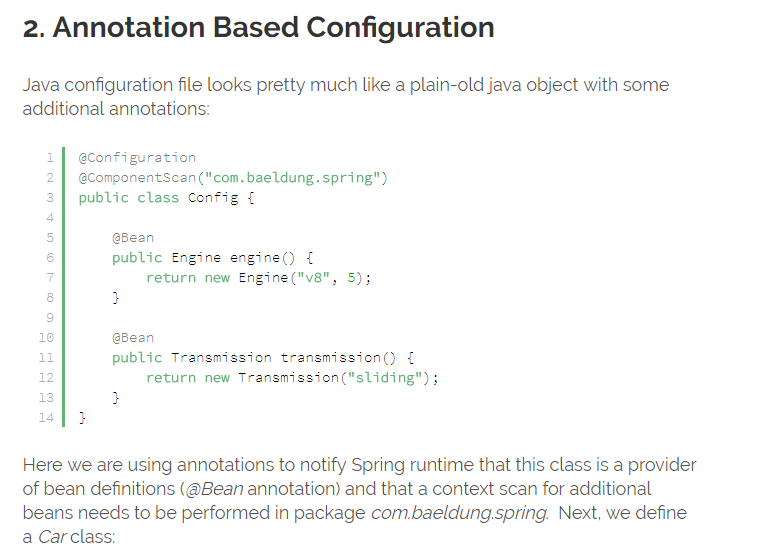
Let's see the simple example to inherit the bean.

**Employee.java**

This class contains three properties, three constructor and show() method to display the values.





|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | @Component  public class Car {        @Autowired      public Car(Engine engine, Transmission transmission) {          this.engine = engine;          this.transmission = transmission;      }  } |

Spring will encounter our *Car*class while doing a package scan and will initialize its instance by calling the *@Autowired*annotated constructor.

Instances of *Engine and Transmission*will be obtained by calling *@Bean*annotated methods of the *Config*class.  Finally, we need to bootstrap an *ApplicationContext*using our POJO configuration:

|  |  |
| --- | --- |
| 1  2 | ApplicationContext context = new AnnotationConfigApplicationContext(Config.class);  Car car = context.getBean(Car.class); |