**How to Initialize List type Properties ?**

**steps:**

**step-1:-**

In the spring bean class declare "List" type variable as dependecy.

**Example :**

package com.nareshit.bean;

public class College

{

private List<String> courses;

//required setters and getters

}

**step-2:-**

Make use of Collection configuration elements in bean configuration file.

i,e we can use <list> tag.

<beans>

<bean id="college" class="com.nareshit.bean.College">

<property name="courses">

<list>

<value>ECE</value>

<value>CSE</value>

<value>IT</value>

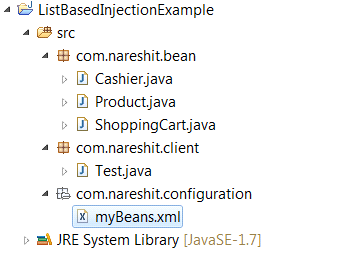
</list>

</property>

</bean>

</beans>

Example2:



**Product.java**

**package** com.nareshit.bean;

**public** **class** Product {

**private** **int** productId;

**private** String productName;

**private** **double** price;

**public** **int** getProductId() {

**return** productId;

}

**public** **void** setProductId(**int** productId) {

**this**.productId = productId;

}

**public** String getProductName() {

**return** productName;

}

**public** **void** setProductName(String productName) {

**this**.productName = productName;

}

**public** **double** getPrice() {

**return** price;

}

**public** **void** setPrice(**double** price) {

**this**.price = price;

}

}

**ShoppingCart.java**

**package** com.nareshit.bean;

**import** java.util.List;

**public** **class** ShoppingCart {

**private** List<Product> products;

**public** List<Product> getProducts() {

**return** products;

}

**public** **void** setProducts(List<Product> products) {

**this**.products = products;

}

}

**Cashier.java**

**package** com.nareshit.bean;

**import** java.io.FileNotFoundException;

**import** java.io.PrintWriter;

**import** java.util.Date;

**import** java.util.List;

**public** **class** Cashier {

**private** **double** totalPrice;

**private** String fileName;

**private** String filePath;

**private** PrintWriter printWriter;

**public** Cashier(String filePath,String fileName){

**this**.filePath=filePath;

**this**.fileName=fileName;

openStream();

}

**public** **void** openStream(){

**try**{

printWriter=**new** PrintWriter(filePath+fileName+".txt");

}**catch**(FileNotFoundException fe){

System.*out*.println("Exception Occured while creating printWriter obj ::"+fe.getMessage());

}

}

**public** **void** calculateTotalPrice(ShoppingCart cart){

List<Product> products=cart.getProducts();

System.*out*.println(products.getClass());

**for**(Product product:products){

writeProductToFile(product);

totalPrice=totalPrice+product.getPrice();

}

printWriter.println("------------------");

printWriter.println("TotalPrice :"+totalPrice);

printWriter.flush();

}

**public** **void** writeProductToFile(Product product){

printWriter.println("productName :"+product.getProductName()+"\tprice:"+product.getPrice()+"\t Date :"+**new** Date());

}

}

**myBeans.xml**

<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"

"<http://www.springframework.org/dtd/spring-beans-2.0.dtd>">

<beans>

<bean id=*"product1"* class=*"com.nareshit.bean.Product"*>

<property name=*"productId"* value=*"4001"*/>

<property name=*"productName"* value=*"keyboard"*/>

<property name=*"price"* value=*"400"*/>

</bean>

<bean id=*"product2”* class=*"com.nareshit.bean.Product"*>

<property name=*"productId"* value=*"4002"*/>

<property name=*"productName"* value=*"mouse"*/>

<property name=*"price"* value=*"200"*/>

</bean>

<bean id=*"product3"* class=*"com.nareshit.bean.Product"*>

<property name=*"productId"* value=*"4003"*/>

<property name=*"productName"* value=*"LCD"*/>

<property name=*"price"* value=*"2000"*/>

</bean>

<bean id=*"shoppingCart"* class=*"com.nareshit.bean.ShoppingCart"*>

<property name=*"products"*>

<list>

<ref bean=*"product1"*/>

<ref bean=*"product2"*/>

<ref bean=*"product3"*/>

</list>

</property>

</bean>

<bean id=*"cashier"* class=*"com.nareshit.bean.Cashier"*>

<constructor-arg index=*"0"* value=*"G://spring/"*/>

<constructor-arg index=*"1"* value=*"productsInformation"*/>

</bean>

</beans>

**Test.java**

package com.nareshit.client;

import org.springframework.beans.factory.xml.XmlBeanFactory;

import org.springframework.core.io.ClassPathResource;

import com.nareshit.bean.Cashier;

import com.nareshit.bean.ShoppingCart;

public class Test {

public static void main(String[] args) {

ClassPathResource resource=

new ClassPathResource("com/nareshit/configuration/myBeans.xml");

XmlBeanFactory factory= new XmlBeanFactory(resource);

ShoppingCart shoppingCart =(ShoppingCart)factory.getBean("shoppingCart");

Cashier cashier=(Cashier)factory.getBean("cashier");

cashier.calculateTotalPrice(shoppingCart);

}

}

**How to initailize Set type properties?**

**step-1:-**

In the spring bean class declare "Set" type variable as dependecy.

**Example :**

package com.nareshit.bean;

public class College{

private Set<String> courses;

//required setters and getters

}

**step-2:-**

Make use of Collection configuration elements in bean configuration file.

i,e we can use <set> tag.

**example:**

<beans>

<bean id="college" class="com.nareshit.bean.College">

<property name="courses">

<set>

<value>ECE</value>

<value>CSE</value>

<value>IT</value>

</set>

</property>

</bean>

</beans>

**How to initialize the Map type properties?**

**Step1** :-In the Spring bean class declare java.util.Map type variable as dependecy.

**Step 2:-** make the use of <map> tag in Spring configuration file

**Example:-**

package com.nareshit;

public class QuestionBean{

private String question;

private Map<String,String> answers;

//setters and getters

}

**xml file:**

<beans>

<bean id="questionBean" class="com.nareshit.QuestionBean">

<property name="question" value="what is java"/>

<property name="answers">

<map>

<entry key="Java is an open source" value="Ajay"></entry>

<entry key="Java is a Platform independent" value="Vijay"></entry>

<entry key="Java is Object Oriented" value="Sanjay"></entry>

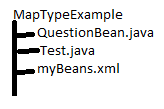
</map>

</property>

</bean>

</beans>

**Initializing Map Object(Where Map Contains String Object Types)**



**QuestionBean.java**

package com.nareshit;

import java.util.Map;

public class QuestionBean {

private String question;

private Map<String,String> answers;

public QuestionBean(String question,Map<String, String> answers) {

this.question=question;

this.answers = answers;

}

//getter methods

}

**myBeans.xml**

<beans>

<bean id="questionBean" class="com.nareshit.QuestionBean">

<constructor-arg value="What is Java?" index="0"/>

<constructor-arg index="1">

<map>

<entry key="Java is an open source" value="Ajay"></entry>

<entry key="Java is a Platform independent" value="Vijay"></entry>

<entry key="Java is Object Oriented" value="Sanjay"></entry>

</map>

</constructor-arg>

</bean>

</beans>

**Test.java**

package com.nareshit;

import org.springframework.beans.factory.BeanFactory;

import org.springframework.beans.factory.xml.XmlBeanFactory;

import org.springframework.core.io.ClassPathResource;

import org.springframework.core.io.Resource;

import java.util.\*;

public class Test {

public static void main(String[] args) {

Resource resource=new ClassPathResource("myBeans.xml");

BeanFactory factory=new XmlBeanFactory(resource);

QuestionBean questionBean=(QuestionBean)factory.getBean("questionBean");

System.out.println("Question :"+ questionBean.getQuestion());

Map<String,String> map=questionBean.getAnswers();

Set<String> keys=map.keySet();

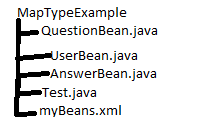
for(String key:keys){  
String value=map.get(key);

System.out.println("Posted By : "+key+" \tAnswer : "+value);  
}

}

}

**Initializing Map Object(Where Map Contains String Object Types)**



**UserBean.java**

package com.nareshit;

public class UserBean {

private String userName,email;

public String toString(){

return " User Name:"+userName+"Email Id:"+email;

}

//required setters and gettes

}

**AnswerBean.java**

package com.nareshit;

import java.util.Date;

public class AnswerBean {

private String answer;

private Date postedDate;

public String toString(){

return " Answer:"+answer+" Posted Date:"+postedDate;

}

//required setters and getters

}

**QuestionBean.java**

package com.nareshit;

import java.util.Map;

public class QuestionBean {

private String question;

private Map<Answer,User> answers;

//setters and getters

}

**applicationContext.xml**

<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"

"<http://www.springframework.org/dtd/spring-beans-2.0.dtd>">

<beans>

<bean id="answerBean1" class="com.nareshit.AnswerBean">

<property name="answer" value="Java is an open source"/>

<property name="postedDate" value="12/21/2015"/>

</bean>

<bean id="answerBean2" class="com.nareshit.AnswerBean">

<property name="answer" value="Java is a Platform independent"></property>

<property name="postedDate" value="12/22/2015"></property>

</bean>

<bean id="userBean1" class="com.nareshit.UserBean">

<property name="userName" value="Vijay Kumar">

</property>

<property name="email" value="Vijay@gmail.com"></property>

</bean>

<bean id="userBean2" class="com.nareshit.UserBean">

<property name="userName" value="Ajay Kumar"></property>

<property name="email" value="Ajay@gmail.com"></property>

</bean>

<bean id="questionBean" class="com.nareshit.QuestionBean">

<property name="question" value="What is Java?"></property>

<property name="answers">

<map>

<entry key-ref="answerBean1" value-ref="userBean1"></entry>

<entry key-ref="answerBean2" value-ref="userBean2"></entry>

</map>

</property>

</bean>

</beans**>**

**Test.java**

package com.nareshit.client;

import org.springframework.beans.factory.BeanFactory;

import org.springframework.beans.factory.xml.XmlBeanFactory;

import org.springframework.core.io.ClassPathResource;

import org.springframework.core.io.Resource;

import java.util.\*;

public class Test {

public static void main(String[] args) {

Resource resource=new ClassPathResource("applicationContext.xml");

BeanFactory factory=new XmlBeanFactory(resource);

QuestionBean questionBean=(QuestionBean)factory.getBean("questionBean");

System.out.println("Question :"+ questionBean.getQuestion());

Map<AnswerBean,UserBean> map=questionBean.getAnswers();

Set<AnswerBean> keys=map.keySet();

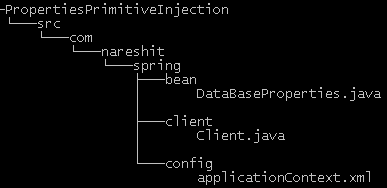
for(AnswerBean answerBean:keys){  
UserBean userBean=map.get(answerBean);

System.out.println("Posted By : "+userBean+" \tAnswer : "+answerBean);  
}

}

}

**Initializing java.util.Properties Object**



**DataBaseProperties.java**

package com.nareshit.spring.bean;

import java.util.Properties;

import java.util.Enumeration;

public class DataBaseProperties {

private Properties dbProperties;

public Properties getDbProperties() {

return dbProperties;

}

public void setDbProperties(Properties dbProperties) {

this.dbProperties = dbProperties;

}

public void displayDetails(){

Enumeration enumeration=dbProperties.keys();

while(enumeration.hasMoreElements()){

String key=(String)enumeration.nextElement();

String name=dbProperties.getProperty(key);

System.out.println(key+"--->"+name);

}

}

}

**applicationContext.xml**

<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="databaseProperties"

class="com.nareshit.spring.bean.DataBaseProperties">

<property name="dbProperties">

<props>

<prop key="driverClass">

oracle.jdbc.driver.OracleDriver</prop>

<prop key="userName">system</prop>

<prop key="password">sathish</prop>

<prop key="url">

jdbc:oracle:thin:@localhost:1521:XE"</prop>

</props>

</property>

</bean>

</beans>

**Client.java**

package com.nareshit.spring.client;

import org.springframework.beans.factory.BeanFactory;

import org.springframework.beans.factory.xml.XmlBeanFactory;

import org.springframework.core.io.ClassPathResource;

import org.springframework.core.io.Resource;

import com.nareshit.spring.bean.DataBaseProperties;

public class Client {

public static void main(String[] args) {

Resource resource=new ClassPathResource("com/nareshit/spring/config/applicationContext.xml");

BeanFactory factory=new XmlBeanFactory(resource);

DataBaseProperties bean=(DataBaseProperties)factory.getBean("databaseProperties ");

bean.displayDetails();

}

}

**what is parent and child containers in Spring ?**

* Spring supports setting parent-child relationship between two IOC containers
* If we have two beanFactory containers in the Application, we can set one BeanFactory into Another beanFactory to allow the beans in one bean factory to refer to the beans of other factory.in this we can declare one beanfactory as a parent and other as a child. This is similar to concept of base class and Derived classes.Derived Class can access the properties of base class,but base classs can not access the properties of derived class.

In order to refer to parent beans, we can use tag **<ref parent=“ “/>**

Apart from **parent** attribute it has **local** which indicates refer to the local bean.Along with it we have **bean** attribute as well,which indicates look in local first if not found then search in parent factory and peform Injection.

**DAOBean.java**

**package** com.nareshit;

**public** **class** DAOBean {

**public** **void** daoMethod(){

System.*out*.println("daoMethod");

}

}

ServiceBean.java

**package** com.nareshit;

**public** **class** ServiceBean {

**private** DAOBean daoBean;

//setters and getters

**public** **void** serviceMethod(){

System.*out*.println("serviceMethod");

daoBean.daoMethod();

}

}

**dao-beans.xml**

<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"

"<http://www.springframework.org/dtd/spring-beans-2.0.dtd>">

<beans>

<bean id=*"daoBean"* class=*"com.nareshit.DAOBean"*/>

</beans>

**service-beans.xml**

<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"

"<http://www.springframework.org/dtd/spring-beans-2.0.dtd>">

<beans>

<bean id=*"serviceBean"* class=*"com.nareshit.ServiceBean"*>

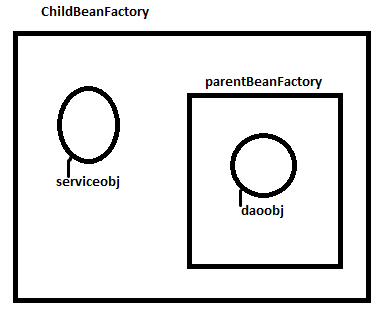
<property name=*"daoBean"*>

<ref bean=”daoBean”/> (OR) <ref parent="daoBean"/>

</property>

</bean>

</beans>



**Test.java**

**package** com.nareshit;

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

**import** org.springframework.beans.factory.xml.XmlBeanFactory;

**import** org.springframework.core.io.ClassPathResource;

**import** org.springframework.core.io.Resource;

**public** **class** Test {

**public** **static** **void** main(String[] args) {

Resource parentResource = **new** ClassPathResource(

"com/nareshit/dao-beans.xml");

BeanFactory parentFactory = **new** XmlBeanFactory(parentResource);

Resource childResource = **new** ClassPathResource(

"com/nareshit/service-beans.xml");

BeanFactory childFactory = **new** XmlBeanFactory(childResource,

parentFactory);

ServiceBean servicebean

= childFactory.getBean("serviceBean", ServiceBean.**class**);

serviceBean.serviceMethod();

}

}

**ApplicationContext container:-**

It is an enhancement for BeanFactory Container. To activate Application Context Container we can create object for the Java class that implements ApplicationContext interface. This is sub interface of BeanFactory(Interface).

There are four important implementation classes for ApplicationContext(Interface).

They are

* FileSystemXmlApplicationContext(C)
* ClassPathXmlApplicatinContext(C)
* XmlWebApplicationcontext(C)
* AnnotationConfigApplicationContext(C) 3.0

**Note:** ApplicationContext container is originally part of Spring context/spring JEE module.

Application Context Container can perform every activity of BeanFactory container but it also gives the following additional features.

1. Ability to work with multiple Spring cfg files.

2. Ability to work with properties files.

3. Support to work with I18n(Internationalization)

4. this container creats the spring bean objects whileloading the spring cfg file.

By activating BeanFactory container for one time we can’t make that container dealing multiple Spring configuration files but this activity is possible by using ApplicationContext Container.

**By Using FileSystemXmlApplicationContext:-**

ApplicationContext context=new FileSystemXmlApplicationContext("D://spring/SpringExample/src/com/nareshit/cfg/myBeans.xml");

If we use FileSystemXmlApplicationContext container ,we need to pass system

Path of the spring configuration file Soin the future if the project location is

Changes we need to change our client application code also .

It is not recomned to use FileSystemXmlApplicationContext class.

**By Using ClasspathXmlApplicationContext**

ApplicationContext context= new ClassPathXmlApplicationContext("com/nareshit/cfg/myBeans.xml");

In the above code container reading the configuration file From class path,but we are not specifying the complete path(System path)

Even if we change the location of the Project in the system our application will work

Without any modifications. Because it reads the configuration file from classpath.

So it isrecomnded to use ClassPathXmlApplicationContext class to create ApplicationContext container.

**By Using XmlWebApplicationContext:-**

This class is used by Spring webmvc module.This container object created

By web container internally,we no need to create our own.

**Key notes on ApplicationContext:-**

It loads spring beans configured in spring configuration file. And manages

The Life cycle of the Spring bean as and when container starts. It won’t wait until getBean() is called.

ApplicationContext container is generally used in Enterprise based applications.

**Auto wiring:-**

Auto wiring means automatically injecting the dependencies.(implict DependecyInjection)

Insead of manually configuring(Explict DIO) the injection we can done automatically by using autowiring.

To implement autowiring we can use “autowire” attribute in <bean> tag configuration.

To perform autowiring for a particular bean we can use any one of the following values

1)byName

2)byType

3)contructor

4)autodetect

1. by using autowire attribute Auto wiring is possible only on reference type bean properties that means auto wiring is not possible on simple, Collection type properties.

2. Autowire attribute decreases the readability of spring cfg file.

**byName:-**

In the case of autowire="byName" the spring container mainly checks for three conditions if all these are valid then it injects the values by setter approach

1.Dependency bean name

2.configured bean name

3.setter method name

-> if dependency name is abc,bean configuration should be abc and the setter Name method should i,e. setAbc(Abc abc)

when it finds autowire="byName" for any bean configuration ,then it first checks for dependency bean name in the dependent bean.

Then it will checks weather any bean is configured in the spring cfg file with the same name.

if it finds then it will call corresponding setterMethod of dependent bean.

**byType:-**

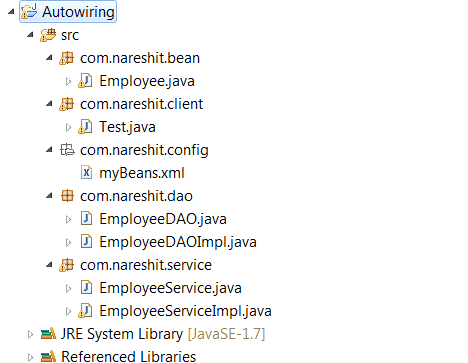
In the case of autowire="byType" the spring container mainly checks for theree conditions if all these are valid then it injects the values by setter appraoch

1.Dependency beanType

2.configured bean Type

3.setter method parameter Type

when it finds "autowire="byType" for any bean configuration ,then it first checks for dependency bean Type in the dependent bean then after it will checks weather any bean is configured in the spring cfg file with the same Type. if it finds then it will call corresponding setterMethod .



**EmployeeService.java**

**package** com.nareshit.service;

**import** com.nareshit.bean.Employee;

**public** **interface** EmployeeService {

public Employee searchEmployee(**int** empNo);

}

**EmployeeServiceImpl.java**

**package** com.nareshit.service;

**import** com.nareshit.bean.Employee;

**import** com.nareshit.dao.EmployeeDAO;

**public** **class** EmployeeServiceImpl **implements** EmployeeService {

**private** EmployeeDAO employeeDao;

**public** **void** setEmployeeDao(EmployeeDAO employeeDao) {

**this**.employeeDao = employeeDao;

}

**public** Employee searchEmployee(**int** empNo) {

Employee emp = employeeDao.searchEmployee(empNo);

**return** emp;

}

}

**EmployeeDAO.java**

**package** com.nareshit.dao;

**import** com.nareshit.bean.Employee;

**public** **interface** EmployeeDAO {

**public** Employee searchEmployee(**int** empNo);

}

**EmployeeDAOImpl.java**

package com.nareshit.dao;

import java.sql.Connection;

import java.sql.PreparedStatement;

import java.sql.ResultSet;

import java.sql.SQLException;

import javax.sql.DataSource;

import com.nareshit.bean.Employee;

public class EmployeeDAOImpl implements EmployeeDAO {

private DataSource dataSource;

public void setDataSource(DataSource dataSource) {

this.dataSource = dataSource;

}

public Employee searchEmployee(int empNo) {

Employee emp = null;

Connection con = null;

try {

con = dataSource.getConnection();

String sql = "select \*from employee where empNo=?";

PreparedStatement pst = con.prepareStatement(sql);

pst.setInt(1, empNo);

ResultSet rs = pst.executeQuery();

if (rs.next()) {

emp = new Employee();

emp.setEmpNo(rs.getInt("empNo"));

emp.setName(rs.getString("name"));

emp.setSalary(rs.getDouble("salary"));

}

} catch (SQLException sqlException) {

System.out.println("Exception Occured while searching Employee() ::"+sqlException.getMessage());

} finally {

if (con != null) {

try {

con.close();

} catch (SQLException sqlException) {

System.out.println("Exception OCcured while closing the connection ::"+

sqlException.getMessage());

}

}

}

return emp;

}

}

**myBeans.xml**

<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" HYPERLINK "http://www.springframework.org/schema/beans/spring-beans-3.0.xsd%22%3E%3Cbeanid=%22dataSource"> HYPERLINK "http://www.springframework.org/schema/beans/spring-beans-3.0.xsd%22%3E%3Cbeanid=%22dataSource"< HYPERLINK "http://www.springframework.org/schema/beans/spring-beans-3.0.xsd%22%3E%3Cbeanid=%22dataSource"bean HYPERLINK "http://www.springframework.org/schema/beans/spring-beans-3.0.xsd%22%3E%3Cbeanid=%22dataSource"id HYPERLINK "http://www.springframework.org/schema/beans/spring-beans-3.0.xsd%22%3E%3Cbeanid=%22dataSource"= *HYPERLINK "http://www.springframework.org/schema/beans/spring-beans-3.0.xsd%22%3E%3Cbeanid=%22dataSource""dataSource*](http://www.springframework.org/schema/beans/spring-beans-3.0.xsd%22%3E%3Cbeanid=%22dataSource)*"*

class=*"org.springframework.jdbc.datasource.DriverManagerDataSource"*>

<property name=*"driverClassName"* value=*"com.mysql.jdbc.Driver"*/>

<property name=*"url"* value=*"jdbc:mysql://localhost:3306/nit"*/>

<property name=*"username"* value=*"root"*/>

<property name=*"password"* value=*"root"*/>

</bean>

<bean id=*"empDao"*

class=*"com.nareshit.dao.EmployeeDAOImpl"* **autowire=*"byName"*>**

</bean>

<bean id=*"empService"*

class=*"com.nareshit.service.EmployeeServiceImpl"* **autowire=*"byType"***>

</bean>

</beans>

**Employee.java**

**package** com.nareshit.bean;

**import** java.io.Serializable;

**public** **class** Employee **implements** Serializable {

**private** **int** empNo;

**private** String name;

**private** **double** salary;

//required setters and getters

}

**Test.java**

package com.nareshit.client;

import java.util.List;

import org.springframework.beans.factory.xml.XmlBeanFactory;

import org.springframework.core.io.ClassPathResource;

import com.nareshit.bean.Employee;

import com.nareshit.service.EmployeeService;

public class Test {

public static void main(String[] args) {

String configFile = "com/nareshit/config/myBeans.xml";

ClassPathResource resource = new ClassPathResource(configFile);

XmlBeanFactory factory = new XmlBeanFactory(resource);

EmployeeService empService = (EmployeeService) factory

.getBean("empService");

System.out.println("Testing of searchEmployee(-) :");

Employee emp = empService.searchEmployee(1004);

if (emp != null) {

System.out.println("Emp No :" + emp.getEmpNo());

System.out.println("Name :" + emp.getName());

System.out.println("Salary :" + emp.getSalary());

} else {

System.out.println("Emp Not Found");

}

}

}

**constructor:-**

In the case of autowire="constructor" the spring container mainly checks for theree

conditions if all these are valid then it injects the values by constructor approach.

1.Dependency beanType

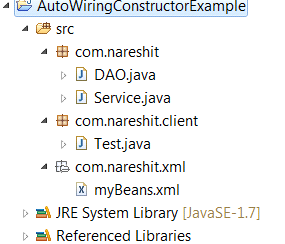
2.configured bean Type

3.constructor parameter Type

when it finds "autowire="constructor" for any bean configuration ,then it first checks for dependency bean Type in the dependent bean then it will checks spring cfg file

weather any bean is confiured with the dependency type,if it finds it will check for constructor

which will takes dependency type as an argument.then will call corresponding constructor.



**Service.java**

**package** com.nareshit;

**public** **class** Service {

**private** DAO dao;

**public** Service(DAO dao){

**this**.dao=dao;

}

**public** **void** serviceMethod(){

System.*out*.println("Service Method");

dao.daoMethod();

}

}

**DAO.java**

**package** com.nareshit;

**public** **class** DAO {

**public** **void** daoMethod(){

System.*out*.println("daoMethod");

}

}

**myBeans.xml**

<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" HYPERLINK "http://www.springframework.org/schema/beans/spring-beans-3.0.xsd%22%3E%3Cbean"> HYPERLINK "http://www.springframework.org/schema/beans/spring-beans-3.0.xsd%22%3E%3Cbean"< HYPERLINK "http://www.springframework.org/schema/beans/spring-beans-3.0.xsd%22%3E%3Cbean"bean](http://www.springframework.org/schema/beans/spring-beans-3.0.xsd%22%3E%3Cbean) id=*"serviceObj"* class=*"com.nareshit.Service"* **autowire=*"constructor"*>**

</bean>

<bean id=*"daoObj"* class=*"com.nareshit.DAO"*/>

</beans>

**Test.java**

package com.nareshit.client;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.nareshit.Service;

public class Test {

public static void main(String[] args){

ApplicationContext context=new

ClassPathXmlApplicationContext("com/nareshit/xml/myBeans.xml");

Service service=(Service)context.getBean("serviceObj");

service.serviceMethod();

}

}

**autodetect:-**

autodetect either using "constructor"(or) byType rules .

always auto-detect first gives the preference to constructor rules

If only zero-arg constructor is found(that means constructor rules are not satisfied ) then "byType" rules applied.

autodetect attributed removed in spring-3.0 xsd so if we are using

spring-3.0. xsd not possible to work with autodetect.

**Note :-** autowire=”no” ==> there is no autowiring

**Global default-autowiring**

Instead of defining autowire attribute for every bean configuration we can set a default-autowire attribute in the<beans> root element to force the all beans declared within <beans> root element to apply this rule.However this root default mode will be overridden by a bean’s own mode if it specified

<beans......... default-autowire="byType">

(or)byName

(or)constructor

(or)autodetect

**mybeans.xml**

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

<beans **default-autowire=”byName”**

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="service" class="com.nareshit.Service"/>

<bean id="dao " class="com.nareshit.DAO" >

</bean>

</beans>

\*\*\* the default value of autowire attribute is "default".

**Dependency-checking:-**

It is used to check all dependencies of bean that are configured via injection are injected or not.

To implement dependency checking to a spring bean we make use of 'dependency-check' attribute of <bean> tag.

this attribute takes any one of the four following values

1) none :- it won't check wheter dependencies injected or not

2) simple :-it checks simple type dependencies injected or not

3) objects:- it checks all the object type dependencies injected or not

4) all:- it checks all types of dependencies injected or not

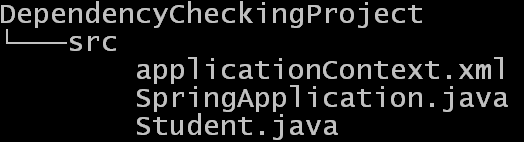
In case if we forgot to set any value either

Simple (or) object type it raise an Exception **like "UnSatisfiedDependencyException"**

Note :

By default a spring beans "dependency-check" value is "default"

To change this default nature of all the beans of the applicationContext we use "default-dependency-check" attribute of<beans> tag.



**Student.java**

public class Student {

private int regno;

private String name,course;

private float fees;

public int getRegno() {

return regno;

}

public void setRegno(int regno) {

this.regno = regno;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public String getCourse() {

return course;

}

public void setCourse(String course) {

this.course = course;

}

public float getFees() {

return fees;

}

public void setFees(float fees) {

this.fees = fees;

}

public void displayStudentDetails(){

System.out.println("Reg No:"+regno);

System.out.println("Name :"+name);

System.out.println("Course :"+course);

System.out.println("Fees :"+fees);

}

}

**applicationContext.xml**

<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-2.5.xsd>">

<bean id="std" class="Student" **dependency-check="simple">**

<property name="course" value="c"/>

<property name="fees" value="500"/>

<property name="name" value="xyz"/>

</bean>

</beans>

**SpringApplication.java**

import org.springframework.beans.factory.BeanFactory;

import org.springframework.beans.factory.xml.XmlBeanFactory;

import org.springframework.core.io.ClassPathResource;

import org.springframework.core.io.Resource;

public class SpringApplication {

public static void main(String[] args) {

Resource resource=new ClassPathResource("applicationContext.xml");

BeanFactory factory=new XmlBeanFactory(resource);

Student std=(Student)factory.getBean("std");

std.displayStudentDetails();

}

}

|  |
| --- |
| Note:-  In the above Example We will get the Following Exception  org.springframework.beans.factory.UnsatisfiedDependencyException:  Error creating bean with name 'std'  defined in class path resource [applicationContext.xml]:  Unsatisfied dependency expressed through bean property 'regno':  Set this property value or disable dependency checking for this bean. |

**Global default-dependency checking**

Explicitly define the dependency checking mode for every beans is tedious and error prone,you can set a default-dependency-check attribute in the<beans> root element to force the entire beans declared within <beans> root element to apply this rule.However this root default mode will be overridden by a bean’s own mode if it specified

<**beans default-dependency-check=”all”**

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-2.5.xsd>">

<bean id="std" class="Student" **>**

<property name="course" value="c"/>

<property name="fees" value="500"/>

<property name="name" value="xyz"/>

</bean>

</beans>

**Spring Annotation Support :-**

Spring 2.0 has a very little support to annotation, when it comes to spring 2.5 , it has extended it's framework to support various aspects of spring development using annotations. In spring 3.0 it started supporting java config style annotations as well.

In Spring 2.0 it has added @Repository and @Required annotations.

In Spring 2.5 it has added few more annotations @Autowired,@Qualifier and @Scope.

along with above annotation's in spring 2.5 sterotype (OR) component type annotations @Component,@Controller, @Service .

In Spring 3.0 few more annotations has been added like @Lazy,@Bean,@Configuration,@DependsOn,@Value etc..

In addition to spring based metadata annotation support,it has started adoption JSR-250 java config project annotations like @PostConstruct,@PreDestory,@Resource ,@Inject and @Named etc...

**@Required annotation :-**  The Spring 2.0 has introduced an annotation @Required . By using this annotation you can make , a particular property has been set with value (OR) not. in spring 2.5 @Required annotation and dependency -check attributed completly removed.

**@Autowired** :- annotation to autowire bean on the setter method, constructor(OR) a filed.

**Example :-**

**EmployeeService.java**

public class EmployeeService{

@Autowired

private EmployeeDAO empDao;

}

**EmployeeDAO.java**

public class EmployeeDAO{

@Autowired  
private DataSource dataSource;

}

The @Autowired Annotation is highly flexible and powerful,and better than <autowire> attribute in bean configuration file.

About the @Autowired annotation to give an intimation to spring IOC container we can use

<context:annotation-config/> in spring file.

**Note :**

By default ,the @Autowired will perform the dependency checking to make sure the property has been wired properly. When Spring container can't find a matching bean to wire, it will throw an Exception. if you want to disable the dependency checking feature then we can use the "required" attribute of @Autowired annotation. if the required attribute value is false then the dependency-checking is disabled. else dependency -checking is enabled. the default value of required attrbute is true**.**

**Example :-**

public class EmployeeService{

@Autowired(required=false)

private EmployeeDAO empDao;

}

**@Qualifier:-**

**for example,In bean configuration file for EmployeeDAO bean two configuration's are there.**

<beans>

<bean id="employeeDao1" class="com.nareshit.dao.EmployeeDAO"/>

<bean id="employeeDao2" class="com.nareshit.dao.EmployeeDAO"/>

</beans>

**In EmployeeService class**

public EmployeeService{

@Autowired

private EmployeeDAO employeeDao;

}

here employeeDao1(OR) employeeDao2 which bean is required to inject don’t know by the container so Container Will rise An Exception. To Solve this problem and to inject a particular bean i,e employeeDao1 (OR) employeeDao2 we can use @Qualifier annotation

The @Qualifier annotation is used to control which bean should be autowire on a field.

**Example:-**

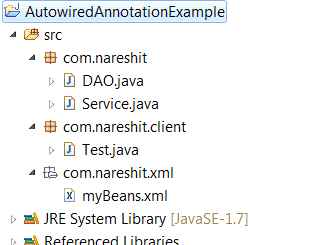
public class EmployeeService{

@Autowired

@Qualifier("employeeDao1")

private EmployeeDAO empDao;

}



**DAO.java**

**package** com.nareshit;

**public** **class** DAO{

**public** **void** daoMethod(){

System.*out*.println("DAOMethod");

}

}

**Service.java**

**package** com.nareshit;

**import** org.springframework.beans.factory.annotation.Autowired;

**public** **class** Service {

@Autowired

**private** DAO dao;

**public** **void** serviceMethod() {

System.*out*.println("serviceMethod");

dao.daoMethod();

}

}

**myBeans.xml**

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:context=*"http://www.springframework.org/schema/context"*

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*](http://www.springframework.org/schema/beans/spring-beans-3.0.xsd)

[*http://www.springframework.org/schema/context*](http://www.springframework.org/schema/context)

[*http://www.springframework.org/schema/context/spring-context-3.0.xsd*](http://www.springframework.org/schema/context/spring-context-3.0.xsd)*"*>

<context:annotation-config />

<bean id=*"dao"* class=*"com.nareshit.DAO"* />

<bean id=*"service"* class=*"com.nareshit.Service"* />

</beans>

**Test.java**

package com.nareshit.client;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.nareshit.Service;

public class Test{

public static void main(String[] args){

ApplicationContext context=

new ClassPathXmlApplicationContext("com/nareshit/xml/myBeans.xml");

Service service=(Service)context.getBean("service");

service.serviceMethod();

}

}

**Stereotype Annotations** :

In Spring there are 4 Stereotype Annotations

**1)@Component :** It makes a java class as an auto scan component.

**2) @Service** – Indicates a Service component in the business layer/service Layer.

**3) @Repository** – Indicates DAO component in the persistence layer.

**4) @Controller** – Indicates a controller component in the Controller layer.

**Example 1 :-**

@Component

public class EmployeeService{

}

**Example 2 :-**

@Component(“empService”)

public class EmployeeService{

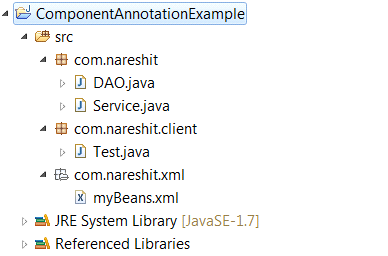
}

@Component Annotation is indicating to the container EmployeeService is class is an auto scan component.

The autoscan components are automatically scanning and Instating by spring ioc container

**About the** Stereotype annotation to give an intimation to the container we can use

<context:component-scan base-package=”com.nareshit”/> in spring configuration file.



**DAO.java**

**package** com.nareshit;

**import** org.springframework.stereotype.Component;

@Component

**public** **class** DAO{

**public** **void** daoMethod(){

System.*out*.println("DAOMethod");

}

}

**Service.java**

**package** com.nareshit;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.stereotype.Component;

@Component("service")

**public** **class** Service {

@Autowired

**private** DAO dao;

**public** **void** serviceMethod() {

System.*out*.println("serviceMethod");

dao.daoMethod();

}

}

**myBeans.xml**

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:context=*"http://www.springframework.org/schema/context"*

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*](http://www.springframework.org/schema/beans/spring-beans-3.0.xsd)

[*http://www.springframework.org/schema/context*](http://www.springframework.org/schema/context)

[*http://www.springframework.org/schema/context/spring-context-3.0.xsd*](http://www.springframework.org/schema/context/spring-context-3.0.xsd)*"*>

<context:component-scan base-package=*"com.nareshit"*>

</context:component-scan>

</beans>

**Test.java**

package com.nareshit.client;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.nareshit.Service;

public class Test{

public static void main(String[] args){

ApplicationContext context=

new ClassPathXmlApplicationContext("com/nareshit/xml/myBeans.xml");

Service service=(Service)context.getBean("service");

service.serviceMethod();

}

}

The @Repository, @Service and @Controller are annotated with @Component annotation internally. the Three annotation's also component Type annotations.

to increase the readability of the program we can use @Service annotation in Service Layer

@Repository annotation in DAO/Persistence Layer and @Contoller annotation in Controller Layer Instead of @Component type annotation.

**Example :**

**EmployeeService.java**

@Service

public class EmployeeService{

}

**EmployeeDAO.java**

@Repository

public class EmployeeDAO {

}

**@Value Annotation**

@Value Annotation is given spring 3.0 version. It is used to inject the values on simple type dependencies

public class CustomerBean{

@Value(“1001”)  
private int customerId;

@Value(“sathish”)

private String customerName;

}

**@Inject :-**

@Inject is J2ee specific annotation,used for Injectiong/autowiring one class into another.

This is similar to @Autowired spring annotation. But the difference between @Autowired supports required attribute where @Inject doesn't has it. @Inject also injects a bean into another bean as similar to @Autowired. it is present in javax.inject package.

**Example :**

**EmployeeService.java**

public class EmployeeService{

@Inject

private EmployeeDAO empDao;

}

**EmployeeDAO.java**

public class EmployeeDAO{

@Inject

private DataSource dataSource;

**}**

**@Named** it is a J2EE annotation and It is similar to @Qualifier spring Annotation

**@Resource** is a J2EE annotation similar to @Inject but it will perform the injection by using byName rules.

**.Properties Files :**

.properties file a text based file it is used to maintain the string type data in key and value format.If we perform any modifications in .java file recompilation and restaring of server system is required.But if perform any modification in .properties file not required to recompile the application and not required to restart the server.

In general the sms gateway details ,database connection details and server ip address and port numbers we will maintain in .properties files .

When we configure the beans in the configuration file, you must remember that it’s not a good practice to mix deployment details, such as the file path, server address, username, and password, with your bean configurations. Generally, the bean configurations are written by application developers while the deployment details are matters for the deployers or system administrators.

Spring f.w provides  [Property HYPERLINK "http://static.springsource.org/spring/docs/3.2.x/javadoc-api/org/springframework/context/support/PropertySourcesPlaceholderConfigurer.html"HYPERLINK "http://static.springsource.org/spring/docs/3.2.x/javadoc-api/org/springframework/context/support/PropertySourcesPlaceholderConfigurer.html" HYPERLINK "http://static.springsource.org/spring/docs/3.2.x/javadoc-api/org/springframework/context/support/PropertySourcesPlaceholderConfigurer.html"PlaceholderConfigurer](http://static.springsource.org/spring/docs/3.2.x/javadoc-api/org/springframework/context/support/PropertySourcesPlaceholderConfigurer.html) class to load properties file data .

PropertyPlaceholderConfigurer resolves ${key} placeholders against [local](http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/core/io/support/PropertiesLoaderSupport.html) [properties](http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/core/io/support/PropertiesLoaderSupport.html) and/or system properties and environment variables.

You can use location (OR) locations property of the PropertyPlaceholderConfigurer to specify the properties file path.

**Example :**

Create a properties file (database.properties), include your database details, put it into your project class path.

db.driverClassName=oracle.jdbc.driver.OracleDriver

db.url=jdbc:oracle:thin:@localhost: 1521:XE

db.username=system

db.password=manager

Now Declare a **PropertyPlaceholderConfigurer** in bean configuration file and map to the ‘database.properties‘ properties file you created just now.

**Example :**

<beans ...>

...

<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">

<property name="location">

<value>database.properties</value>

</property>

</bean>

<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">

<property name="driverClassName" value="${db.driverClassName}" />

<property name="url" value="${db.url}" />

<property name="username" value="${db.username}" />

<property name="password" value="${db.password}" />

</bean>

</beans>

From Spring 3.x the registration of PropertyPlaceholderConfigurer can be simply through the

<context:property-placeholder> element.

**Example :**

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:context="http://www.springframework.org/schema/context"

xsi:schemaLocation="http://www.springframework.org/schema/beans

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

<http://www.springframework.org/schema/context>

<http://www.springframework.org/schema/context/spring-context-3.0.xsd>">

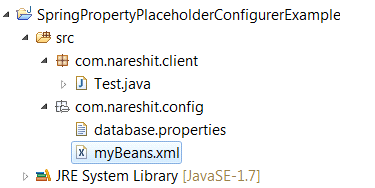
<context:property-placeholder location="database.properties" />

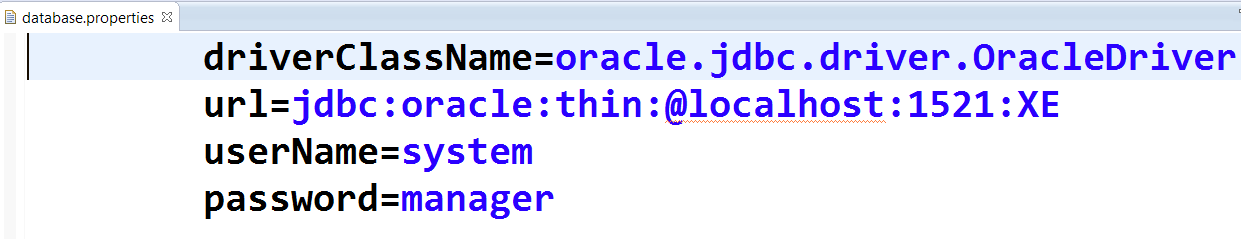
</beans>

**Note :**

BeanFactory container can’t work with properties files. Only Application Context container work with properties files.

**Example :-**





**myBeans.xml**

<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"

"<http://www.springframework.org/dtd/spring-beans-2.0.dtd>">

<beans>

<bean id=*"dataSource"* class=*"org.springframework.jdbc.datasource.DriverManagerDataSource"*>

<property name=*"driverClassName"* value=*"${driverClassName}"* />

<property name=*"url"* value=*"${url}"* />

<property name=*"username"* value=*"${userName}"* />

<property name=*"password"* value=*"${password}"* />

</bean>

<bean class=*"org.springframework.beans.factory.config.PropertyPlaceholderConfigurer"*>

<property name=*"location"*>

<value>com/nareshit/config/database.properties

</value>

</property>

</bean>

</beans>

**Test.java**

package com.nareshit.client;

import java.sql.Connection;

import java.sql.DatabaseMetaData;

import java.sql.SQLException;

import javax.sql.DataSource;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Test {

public static void main(String[] args) throws SQLException {

ApplicationContext context=new

ClassPathXmlApplicationContext("com/nareshit/config/myBeans.xml");

DataSource ds=(DataSource)context.getBean("dataSource");

Connection con=ds.getConnection();

DatabaseMetaData dbmd=con.getMetaData();

System.out.println(dbmd.getDatabaseProductVersion());

}

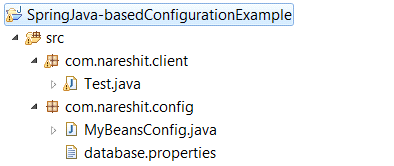
}

**working with @Configuration and @Bean :-**

Instead of declaring a class as spring bean in a configuration file, you can declare it in a class. The class in which you want to provide the configuration about other beans, that class is called configuration class and you need to annotate with @Configuration.

In this class you need to provide the methods which are responsible for creating object's of your bean classes, these methods has to be annotated with @Bean.

**Example :-**



**MyBeansConfig.java**

package com.nareshit.config;

import javax.sql.DataSource;

import org.springframework.beans.factory.annotation.Value;

import org.springframework.beans.factory.config.PropertyPlaceholderConfigurer;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.core.io.ClassPathResource;

import org.springframework.core.io.Resource;

import org.springframework.jdbc.datasource.DriverManagerDataSource;

@Configuration

public class MyBeansConfig{

@Value("${driverClassName}")

private String driverClassName;

@Value("${url}")

private String url;

@Value("${userName}")

private String userName;

@Value("${password}")

private String password;

@Bean(name="dataSource")

public DataSource dataSource(){

DriverManagerDataSource

dataSource=new DriverManagerDataSource();

dataSource.setDriverClassName(driverClassName);

dataSource.setUrl(url);

dataSource.setUsername(userName);

dataSource.setPassword(password);

return dataSource;

}

@Bean

public static PropertyPlaceholderConfigurer placeholderConfigurer(){

PropertyPlaceholderConfigurer placeholderConfigurer=

new PropertyPlaceholderConfigurer();

Resource resource=

new ClassPathResource("com/nareshit/config/database.properties");

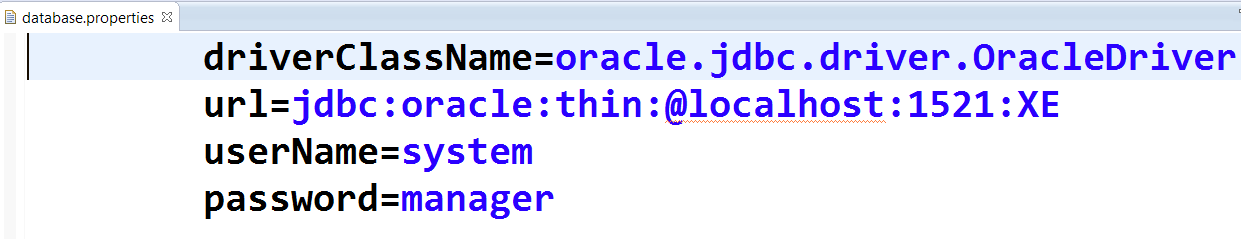
placeholderConfigurer.setLocation(resource);

return placeholderConfigurer;

}

}

**database.properties**



**Test.java**

**package** com.nareshit.client;

**import** java.sql.Connection;

**import** java.sql.DatabaseMetaData;

**import** java.sql.SQLException;

**import** javax.sql.DataSource;

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

**import** org.springframework.context.annotation.AnnotationConfigApplicationContext;

**import** com.nareshit.config.MyBeansConfig;

**public** **class** Test {

**public** **static** **void** main(String[] args) **throws** SQLException {

ApplicationContext context = **new** AnnotationConfigApplicationContext(

MyBeansConfig.**class**);

DataSource ds = (DataSource) context.getBean("dataSource");

Connection con = ds.getConnection();

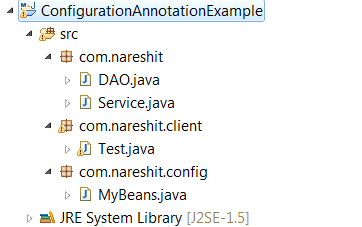
DatabaseMetaData dbmd = con.getMetaData();

System.*out*.println(dbmd.getDatabaseProductVersion());

}

}

**Example2 :-**



**Service.java**

**package** com.nareshit;

**public** **class** Service {

**private** DAO dao;

**public** **void** setDao(DAO dao) {

**this**.dao = dao;

}

**public** **void** serviceMethod() {

System.*out*.println("serviceMethod");

dao.daoMethod();

}

}

**DAO.java**

**package** com.nareshit;

**public** **class** DAO{

**public** **void** daoMethod(){

System.*out*.println("DAOMethod");

}

}

**MyBeans.java**

package com.nareshit.config;

import org.springframework.beans.factory.annotation.Autowire;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import com.nareshit.DAO;

import com.nareshit.Service;

@Configuration

public class MyBeans {

@Bean(name="service",autowire=Autowire.BY\_TYPE)

public Service service(){

return new Service();

}

@Bean(name="dao")

public DAO dao(){

return new DAO();

}

}

**Test.java**

package com.nareshit.client;

import org.springframework.context.ApplicationContext;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.nareshit.Service;

public class Test{

public static void main(String[] args){

ApplicationContext context=

new AnnotationConfigApplicationContext(com.nareshit.config.MyBeans.class);

Service service=(Service)context.getBean("service");

service.serviceMethod();

}

}

**Spring bean scopes :**

In Spring, bean scope is used to decide which type of bean instance should be return from Spring container back to the caller.

5 types of bean scopes supported by spring F.w:

singleton – Return a same bean instance each time when we requested with getBean(-) metho

prototype – Return a new bean instance each time when requested with getBean(-) method

request – Return a single bean instance per HTTP request. \*

session – Return a single bean instance per HTTP session. \*

globalSession – is similar to session. but this scope is used in portlet environment.

The session scope is used in servlet enviornment.

**The default scope of spring bean is singleton.**

**Example :**

package com.nareshit.bean;

public class Student{  
private int studentId;

private String studentName;

//setters and getters

//override toString() method  
}

in myBeans.xml file configure Student class as follows:-

<beans>

<bean id="student" class="com.nareshit.bean.Student" scope="singleton">

</bean>

</beans>

in client application :-

Student std1=container.getBean("student",Student.class);

std1.setStudentId(101);

std1.setStudentName("rama");

System.out.println(std1);//[101 rama]

when we are calling getBean(-) second time also the spring container returns same existed bean object reference because the student bean scope is singleton.

std1=container.getBean("student",Student.class);

System.out.println(std1);//[101 rama]

prototype scope:

in myBeans.xml file configure Student class as follows:-

<beans>

<bean id="student" class="com.nareshit.bean.Student" scope="prototype">

</bean>

</beans>

in client application :-

Student std1=container.getBean("student",Student.class);

std1.setStudentId(101);

std1.setStudentName("rama");

System.out.println(std1);//[101 rama]

when we are calling getBean(-) second time the spring container creates a new student object returns because the student bean scope is prototype.

std1=container.getBean("student",Student.class);

System.out.println(std1);//[0 null] (default values)

**Note:**

spring bean scope="singleton" and singleton design pattern both are different.

spring bean scope="singleton" not makes a spring bean class as a singleton class.

**in myBeans.xml file configure Student class as follows:-**

**<beans>**

**<bean id="student1" class="com.nareshit.bean.Student" scope="singleton"/>**

**<bean id="student2" class="com.nareshit.bean.Student" scope="singleton"/>**

**</beans>**

**wth the above configurtion the spring container creates two student bean object's in the memory.**

**Factory-Methods :**

**if any java method is having the capability of for constructing and returning of some class object reference is called as factory method.**

**There are two factory methods.**

**1)Static factory method**

**2)instance factory method**

we can make the spring container to create spring bean class Object either by using static factory method (OR) instance factory method (OR) regular constructor approach.

**with static Factory-method:-**

in java For some classes object creation is possible only with factory-method's.

For that kind of classes to create object with static-factory-method we can use "factory-method" attribute along with "class" attribute of "bean" tag and give an intimation to spring container.

**Example :-**

<beans>

<bean id="cls" class="java.lang.Class" factory-method="forName">

<constructor-arg value="oracle.jdbc.driver.OracleDriver"/>

</bean>

<bean id="connection" class="java.sql.DriverManager" factory-method="getConnection">

<constructor-arg value="jdbc:oracle:thin:@localhost:1521:XE"/>

<constructor-arg value="system"/>

<constructor-arg value="manager"/>

</bean>

<bean id="statement" factory-bean="connection" factory-method="createStatement"/>

<beans>

**Note :** To specifies the argument values of a factory method use <constructor-arg> tag

**With Instance-Factory method :**

All the classes object’s are cannot be created by using new Operator,few can be constructed by calling factory method on the class,but few objects can be constructed by calling methods on other classes.

So, In order to create such type of Objects you need to instantiate by using Instance-factory methods.

**Note :-**

while configuring spring bean "factory-method" attribute is palces along with "class" attribute then spring container uses static factorymethod for instantiation.

while configuring spring bean if "factory-method" attribute is placed along with "factory-bean" attribute and without class attribute then spring container uses instance-factory method to create spring bean class Object.

**Spring Bean Life Cycle**

In addition to bean creation, the Spring IOC container is also responsible for managing the life cycle of your beans, and it allows you to perform custom tasks at particular points of their life cycle. Your tasks should be encapsulated in life cycle methods for the Spring IOC container to call at a suitable time.

The following list shows the steps through which the Spring IOC container manages the life cycle of a bean. This list will be expanded as more features of the IOC container are introduced.

1.spring container Creates the bean instance either by using a constructor or by a factory method.

2. It the Set the values and bean references to the bean properties.

3. It Calls the initialization lifecycle methods.(callback initialization method)

4. The bean is ready to be used.

5. When the container is shut down, call the destruction lifecycle methods.(callback destruction method)

In Many real-world components have to perform certain types of initialization tasks before they are ready to be used. Such tasks include opening a file, opening a network/database connection, allocating memory, and so on. Also, they have to perform the corresponding destruction tasks at the end of their

life cycle. So, you have a need to customize bean initialization and destruction in the Spring IoC container.

There are three ways that Spring can recognize your initialization and destruction lifecycle methods.(callback methods)

First Approach , your beanclass can implement the InitializingBean and DisposableBean life cycle interfaces and overrides the afterPropertiesSet() and destroy() methods for initialization and destruction.

Second Approach, you can set the **init-method** and **destroy-method**  attributes in the bean declaration and specify the lifecycle method names.

In Spring 2.5 or later, you can also annotate the initialization and destruction

life cycle methods with the life cycle annotations **@PostConstruct** and **@PreDestroy**,

To understand how the Spring IoC container manages the life cycle of your beans, let’s consider an

Example.

**By Implementing the InitializingBean and DisposableBean Interfaces**

Spring allows your bean to perform initialization and destruction tasks in the callback methods

afterPropertiesSet() and destroy() by implementing the InitializingBean and DisposableBean

interfaces. During bean construction, Spring will notice that your bean implements these interfaces and call the callback methods(life cycle methods) at a suitable time.

**public** **class** Cashier **implements** InitializingBean,DisposableBean {

**private** **double** totalPrice=0.0;

**private** String fileName;

**private** String filePath;

**private** PrintWriter printWriter;

**public** **void** setFileName(String fileName) {

**this**.fileName = fileName;

}

**public** **void** setFilePath(String filePath) {

**this**.filePath = filePath;

}

**public** **void** openStream(){

**try**{

printWriter=**new** PrintWriter(filePath+"/"+fileName+".txt");

}

**catch**(FileNotFoundException fnfe){

System.out.println("Exception Occured while executing "

+ "the openStream() ::"+fnfe.getMessage());

}

}

**public** **void** calculateTotalPrice(ShoppingCart cart){

List<Product> list= cart.getProducts();

System.out.println(list.getClass());

**for**(Product product:list){

totalPrice=totalPrice+product.getPrice();

writeProductToFile(product);

}

printWriter.println("================");

printWriter.println("Total Price :"+totalPrice);

printWriter.flush();

}

**public** **void** writeProductToFile(Product product){

printWriter.println(**new** Date()+"\t"+product.getProductName()+"\t"+product.getPrice());

}

**public** **void** closeStream(){

printWriter.close();

}

@Override

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

closeStream();

}

@Override

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

openStream();

}

}

**In spring cfg file:-**

<beans>

<bean id="cashier" class="com.nareshit.domain.Cashier">

<property name="fileName" value="productsInformation"/>

<property name="filePath" value="G://spring/"/>

</bean>

</beans>

**Note :-**

Both BeanFactory and ApplicationContext container support life cycle methods. We can’t stop BeanFactory container explicitly. but it is possible to stop ApplicationContext container explicitly as per the application requirement's.

**Example code :**

ClassPathXmlApplicationContext context=new ClassPathXmlApplicationContext("myBeans.xml");

----

----

----

context.close();

to destroy the spring bean we need to call factory.destroyXxxMethos() at the end of client application When the container is BeanFactory.

**Example code :**

XmlBeanFactory factory=new XmlBeanFactory(resource);

----

----

factory.destroySingletons();

**By using the init-method and destroy-method Attributes**

A better approach of specifying the initialization and destruction callback methods is by setting the

init-method and destroy-method attributes in your bean declaration.

With these two attributes set in the bean declaration, your Cashier class no longer needs to

implement the InitializingBean and DisposableBean interfaces. You can also delete the

afterPropertiesSet() and destroy() methods as well.

In the Cashier class, the openStream() method creates printWriter Stream based given filePath and fileName.

the closeStream() method closes the printWriter stream to release its system resources.

The openStream() and closeStream() methods are declared as callback Initialization and call back destruction methods by using init-method and destroy-method attributes in the bean configuration file as follows.

**In spring cfg file**

<beans>

<bean id="cashier" class="com.nareshit.domain.Cashier"

init-method="openStream" destroy-method="closeStream">

<property name="fileName" value="productsInformation"/>

<property name="filePath" value="G://spring/"/>

</bean>

</beans>

And Cashier class as follows

**Cashier.java**

**public** **class** Cashier {

**private** **double** totalPrice=0.0;

**private** String fileName;

**private** String filePath;

**private** PrintWriter printWriter;

**public** **void** setFileName(String fileName) {

**this**.fileName = fileName;

}

**public** **void** setFilePath(String filePath) {

**this**.filePath = filePath;

}

**public** **void** openStream(){

**try**{

printWriter=**new** PrintWriter(filePath+"/"+fileName+".txt");

}

**catch**(FileNotFoundException fnfe){

System.out.println("Exception Occured while executing "

+ "the openStream() ::"+fnfe.getMessage());

}

}

**public** **void** calculateTotalPrice(ShoppingCart cart){

List<Product> list= cart.getProducts();

System.out.println(list.getClass());

**for**(Product product:list){

totalPrice=totalPrice+product.getPrice();

writeProductToFile(product);

}

printWriter.println("================");

printWriter.println("Total Price :"+totalPrice);

printWriter.flush();

}

**public** **void** writeProductToFile(Product product){

printWriter.println(**new** Date()+"\t"+product.getProductName()+"\t"+product.getPrice());

}

**public** **void** closeStream(){

printWriter.close();

}

}

**By using @PostContruct and @PreDestroy**

public class Cashier{

---

---

---

**@PostContruct**

public void openStream(){  
}

**@PreDestroy**

public void closeStream(){  
}  
}

**Q) What is container call back method (or) Container Life cycle method?**

A) The method that is called by underlying container automatically based on the Event that is raised is called as container callback method (or) container Life cycle method.

**Bean Post Processors**

You would like to register your own plug-ins in the Spring IoC container to process the bean instances during construction.

A bean post processor allows additional bean processing before and after the initialization callback method.

BeanPostProcessor is used to provide some common logic to all the configured beans in Spring cfg file.

Typically, bean post processors are used for checking the validity of bean properties or altering bean properties according to particular criteria.

BeanPostProcessor is used to implement common logic in the project.

The basic requirement of a bean post processor is to implement the BeanPostProcessor interface.

You can process every bean before and after the initialization callback method by implementing the

postProcessBeforeInitialization() and postProcessAfterInitialization() methods.

The Spring IOC Container will pass each bean instance to these two methods before and after calling the initialization callback method, as illustrated in the following list:

1. Spring container Creates the bean instance either by a constructor or by a factory method.

2. It Set the values and bean references to the bean properties.

3. It Calls the setter methods defined in the aware interfaces.

4. It Pass the bean instance to the postProcessBeforeInitialization() method of

each bean post processor.

5. It Calls the initialization callback methods(lifecycle method).

6. It Pass the bean instance to the postProcessAfterInitialization() method of

each bean post processor.

7.The bean is ready to be used state.

8. When the container is shut down, It calls the destruction callback methods(life cycle methods).

When using a bean factory as your IoC container, bean post processors can only be registered

programmatically, or more accurately, via the addBeanPostProcessor() method. However, if you are

using an application context, the registration will be as simple as declaring an instance of the processor

in the bean configuration file, and then it will get registered automatically.

**How It Works**

Suppose you would like check the directory is existed (OR) not before executing Cashier class openStream method(callback Initialization method) in the above example to avoid the FileNotFoundException

We can implement the required logic by using BeanPostProcessor methods.

**Cashier.java**

Cashier Bean is Same as the Above Example

**MyBeanPostProcessor.java**

**public** **class** MyBeanPostProcessor **implements** BeanPostProcessor {

**public** Object postProcessAfterInitialization(Object beanObject,

String beanName) **throws** BeansException {

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

**return** beanObject;

}

**public** Object postProcessBeforeInitialization(Object beanObject,

String beanName) **throws** BeansException {

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

**if** (beanObject **instanceof** Cashier) {

Cashier cashier = (Cashier) beanObject;

String directory = cashier.getFilePath();

File file = **new** File(directory);

**boolean** b = file.mkdirs();

**if** (b == **true**) {

System.out.println("Directory is created ::" + directory);

} **else** {

System.out.println("Directory is already existed ");

}

}

**return** beanObject;

}

}

**In Spring cfg file :**

<beans>

<bean id="cashier" class="com.nareshit.domain.Cashier"

init-method="openStream" destroy-method="closeStream">

<property name="fileName" value="productsInformation"/>

<property name="filePath" value="G://spring/"/>

</bean>

<bean id="myBeanPostProcessor"

class="com.nareshit.bean.MyBeanPostProcessor">

</bean>

</beans>

**Aware Interfaces :**

Sometimes we need Spring Framework objects in our beans to perform some operations, for example reading ServletConfig and ServletContext parameters or to know the bean definitions loaded by the ApplicationContext. That’s why spring framework provides a bunch of \*Aware interfaces that we can implement in our bean classes.

org.springframework.beans.factory.Aware is the root marker interface for all these Aware interfaces. All of the \*Aware interfaces are sub-interfaces of Aware and declare a single setter method to be implemented by the bean. Then spring context uses setter-based dependency injection to inject the corresponding objects in the bean and make it available for our use.

.Some of the important Aware interfaces are:

* **ApplicationContextAware** – to inject ApplicationContext object.
* **BeanFactoryAware** – to inject BeanFactory object.
* **BeanNameAware** – to know the bean name defined in the configuration file.
* **ServletContextAware** – to inject ServletContext object in MVC application, example usage is to read context parameters and attributes.
* **ServletConfigAware** – to inject ServletConfig object in MVC application, example usage is to get servlet config parameters.

Let’s see these Aware interfaces usage in action by implementing few of them in a class that we will configure as spring bean.

**MyAwareService.java**

public class MyAwareService implements ApplicationContextAware, BeanNameAware {

@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);

}

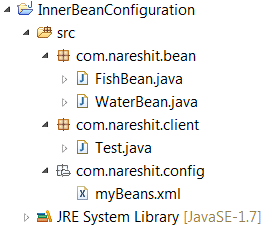
}

**What is InnerBean?**

A spring bean configured with in <property> (OR) <constructor-arg> tag by using <bean> tag is

known as Inner bean configuration while configuring Inner beans name/id attributes

are not use ful. so we can say inner bean also called anonymous bean.



**FishBean.java**

**package** com.nareshit.bean;

**public** **class** FishBean {

**private** String fishName;

**public** String getFishName() {

**return** fishName;

}

**public** **void** setFishName(String fishName) {

**this**.fishName = fishName;

}

}

**WaterBean.java**

**package** com.nareshit.bean;

**public** **class** WaterBean {

**private** FishBean fishBean;

**public** FishBean getFishBean() {

**return** fishBean;

}

**public** **void** setFishBean(FishBean fishBean) {

**this**.fishBean = fishBean;

}

}

**myBeans.xml**

<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"

"<http://www.springframework.org/dtd/spring-beans-2.0.dtd>">

<beans>

<bean id="waterBean" class="com.nareshit.bean.WaterBean">

<property name="fishBean">

<bean class="com.nareshit.bean.FishBean">

<property name="fishName" value="Goldfish"/>

</bean>

</property>

</bean>

</beans>

**Test.java**

**package** com.nareshit.client;

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

**import** org.springframework.context.support.ClassPathXmlApplicationContext;

**import** com.nareshit.bean.FishBean;

**import** com.nareshit.bean.WaterBean;

**public** **class** Test {

**public** **static** **void** main(String[] args) **throws** Exception {

String configFile ="com/nareshit/config/myBeans.xml";

ApplicationContext context=**new** ClassPathXmlApplicationContext(configFile);

WaterBean wb = context.getBean("waterBean", WaterBean.**class**);

FishBean fb = wb.getFishBean();

System.out.println("Fish Name :" + fb.getFishName());

}

}

**How to work with multiple Spring configuration Files?**

when the spring cfg file size huge,it is difficult to maintain ,so in that scenarious we generally have multiple configuration files.

We can work with multiple spring cfg files in two ways.

1. By importing other spring configuration file into the current configuration file.

Example:-

<import resource =”other-spring-beans.xml”/>

2. By passing all spring bean cfg files as a string array to the constructor of the

ApplicationContext interface implementation classes.

ApplicationContext context=new classPathXmlApplicationContext(new String[]{“beans-1.xml”,”beans-2.xml”,”beans-3.xml”});

**What is Bean Inheritance?**

Inheritance is the concept of reusing the existing functionality .

In case of java you can inherit a class fom an interface (OR) another class.when you inherit a class from another class,your child (OR) Derived class get’s all the functionalities of Your base class.

When it comes to spring Bean Inheritance,it talks about how to reuse the existing bean configuration instead of re-defining again. Let’s consider a scenario where your class contains 5 properties , if we want to configure it as a spring bean you need to inject values for 5 properties via constructor (OR) setter injection.

If we want to create 20 beans of that class ,we need to configure for all the 20 beans for setter (OR)constructor injection. In case If most of the properties has same value,even then also we need to re-write the configuration. This leads to duplicate configuration declaration in high amount of maintenance.

In order to avoid this you can declare the configuration in one bean which acts as parent bean.And All the remaining 19 bean**.** Declarations can inherit their declaration values from the parent bean,so that we don’t need to repeatedly wirte the same configuration in all the child beans..In this way if we modify the attribute value in parent bean,it will automatically reflects in all Its 19 child beans.The child bean can override the inherited value of parent by re-declaring at the child level.

**The following example is show s how to use the feature :**

**Person.java**

package com.nareshit.bean;

public class Person {

private String name;

private String phone;

private String email;

private String city;

private String country;

//required setters and getters

**public** String toString() {

**return** "Person [name=" + name + ", phone=" + phone + ", email=" + email+ ", city=" + city + ", country=" + country + "]";

}

}

**myBeans.xml**

<beans>

<bean id="parentPersonBean" class="com.nareshit.bean.Person">

<property name="city" value="Hyd" />

<property name="country" value="India" />

</bean>

<bean id="childPersonBean1" parent="parentPersonBean">

<property name="name" value="Srinu" />

<property name="phone" value="8989898988" />

<property name="email" value="Srinu@nareshit.in" />

</bean>

<bean id="childPersonBean2" parent="parentPersonBean">

<property name="name" value="Hari" />

<property name="phone" value="7875444333" />

<property name="email" value="hari@nareshit.in" />

</bean>

</beans>

In the above cfg file ‘parentPersonBean’ bean contains ‘hyd’ and ‘India’ values for city and country properties , and the ‘childPersonBean1’ bean and ‘childPersonBean2’ bean inherited city and country properties values from its parent (‘parentPersonBean’).

**Test.java**

**package** com.nareshit.client;

**import** org.springframework.context.support.ClassPathXmlApplicationContext;

**import** com.nareshit.bean.Person;

**public** **class** Test {

**public** **static** **void** main(String[] args) {

ClassPathXmlApplicationContext context=**new**

ClassPathXmlApplicationContext("com/nareshit/config/myBeans.xml");

Person childPersonBean1= context.getBean("childPersonBean1",Person.**class**);

Person childPersonBean2= context.getBean("childPersonBean2",Person.**class**);

System.out.println(childPersonBean1);

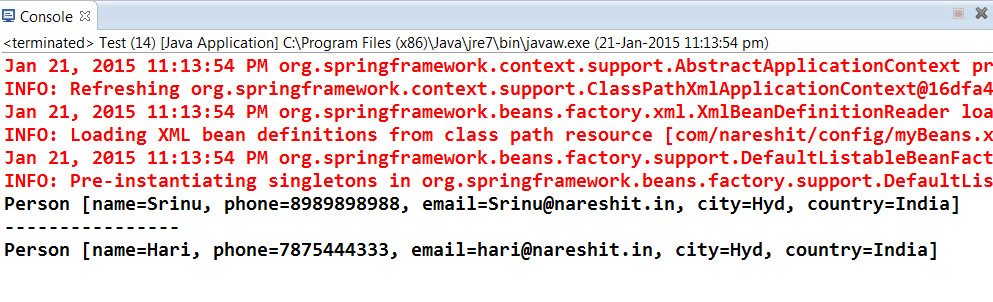
System.out.println("----------------");

System.out.println(childPersonBean2);

}

}

**Output :**



**Inheritance with abstract**

In above example, the ‘parentPersonBean’ is still able to instantiate, for example,

Person parentPersonBean= context.getBean("parentPersonBean",Person.**class**);

If you want to make this base bean as a template and not allow others to instantiate it, you can add an ‘**abstract**‘ attribute in the <bean> element. For example

<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"

"<http://www.springframework.org/dtd/spring-beans-2.0.dtd>">

<beans>

<bean id="parentPersonBean" class="com.nareshit.bean.Person" abstract="true">

<property name="city" value="Hyd" />

<property name="country" value="India" />

</bean>

<bean id="childPersonBean1" parent="parentPersonBean">

<property name="name" value="Srinu" />

<property name="phone" value="8989898988" />

<property name="email" value="Srinu@nareshit.in" />

</bean>

<bean id="childPersonBean2" parent="parentPersonBean">

<property name="name" value="Hari" />

<property name="phone" value="7875444333" />

<property name="email" value="hari@nareshit.in" />

</bean>

</beans>

Now, the ‘parentPersonBean’ bean is a pure template, for bean to inherit it only, if you try to instantiate it, you will encounter the following error message.

**Person parentPersonBean= context.getBean("parentPersonBean",Person.class);**

Exception : Exception in thread "main" org.springframework.beans.factory.BeanIsAbstractException: Error creating bean with name 'parentPersonBean': Bean definition is abstract

**Pure Inheritance Template**

Actually, parent bean is not necessary to define class attribute, often times, you may just need a common property for sharing. Here’s is an example

<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"

"<http://www.springframework.org/dtd/spring-beans-2.0.dtd>">

<beans>

<bean id="parentPersonBean" abstract="true">

<property name="city" value="Hyd" />

<property name="country" value="India" />

</bean>

<bean id="childPersonBean1" parent="parentPersonBean" class="com.nareshit.bean.Person">

<property name="name" value="Srinu" />

<property name="phone" value="8989898988" />

<property name="email" value="Srinu@nareshit.in" />

</bean>

<bean id="childPersonBean2" parent="parentPersonBean" class="com.nareshit.bean.Person">

<property name="name" value="Hari" />

<property name="phone" value="7875444333" />

<property name="email" value="hari@nareshit.in" />

</bean>

</beans>

In this case, the ‘parentPersonBean’ bean is a pure template, to share its ‘city and country’ properties only.

**Overrride it**

However, you are still allow to override the inherited value by specify the new value in the child bean. Let’s see this example

<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"

"<http://www.springframework.org/dtd/spring-beans-2.0.dtd>">

<beans>

<bean id="parentPersonBean" class="com.nareshit.bean.Person" abstract="true">

<property name="city" value="Hyd" />

<property name="country" value="India" />

</bean>

<bean id="childPersonBean1" parent="parentPersonBean" >

<property name="name" value="Srinu" />

<property name="phone" value="8989898988" />

<property name="email" value="Srinu@nareshit.in" />

<property name="city" value="Chennai"/>

</bean>

<bean id="childPersonBean2" parent="parentPersonBean" class="com.nareshit.bean.Person">

<property name="name" value="Hari" />

<property name="phone" value="7875444333" />

<property name="email" value="hari@nareshit.in" />

</bean>

</beans>

**Conclusion**

The Spring bean configuration inheritance is very useful to avoid the repeated common value or configurations for multiple beans.

**What is collection merging?**

* In spring 2.0 the container supports the collection merging. In this your parent bean can declare a list as parent list.In your child beans you can declare a list with values ,you can inherit the values of your parent list values into your child bean list values .As the list merged with parent list values,this is called Collection merging.

**Course.java**

public class Course{  
private List<String> subjects;

//setter and getter  
}

**myBeans.xml**

<beans>

<bean id=“parentCourse” class=“Course” abstract=“true”>

<property name=“subjects”>

<list>

<value>c</value>

<value>java</value>

</list></property>  
<bean id=“childCourse” class=“Course” parent=“parentCourse”>

<property name=“subjects”>

<list merge=“true”>

<value>DS</value>

</list>

</property>

</bean>

</beans>

**Bean Aliasing**

In Spring when you configure a class as bean you will declare an id with which you want to retrieve it back from the container. Along with id you can attach multiple names to beans, and these names act as alias names with which you can look up the bean from the container. To declare multiple names You need to declare an “name” attribute at the bean tag level . We can separate multiple names with comma(OR) with space.

Following Code showing Alias Names for Student bean

<bean id=”student1” name=”student2,student3,student4” class=”com.nareshit.bean.Student”>

</bean>

You can retrieve the above bean with either student1 (OR) student2(OR) student3 names. You can even get all the names of the bean using factory.getAliases(“oneName”);

Id attribute is holding actual name .

name attribute holding alias names.

In general bean aliasing is used for ease maintenance of the configuration.

In Spring 2.0 a new tag has been introduced <alias> using which you can declare multiple names for the bean. The Syntax is as follows

**P-Namespace and C-Namespace:**

* If we want to perform setter injection on a spring bean we need to use <property> tag.Instead of writing length<property> tag declaration under the <bean> tag we can replace with short form of representing the same with p-namespace.
* In order to use the p-namespace,you first need to import the “htttp://www.springframework.org/schema/p” Name space in the spring bean cfg file.

Consider following example:  
  
**Student.java**

package com.nareshit.bean;  
  
public class Student {  
    private String name;  
    private int age;  
    private Course course;  
  
   //required setters and getters      
  
    public void displayInfo(){  
          
        System.out.println("Student Name : "+name);  
        System.out.println("Student Age : "+ age);  
        System.out.println("Course name : "+course.getCourseName());  
    }  
}

**Course.java**

package com.nareshit.bean;  
public class Course {  
  
private     String courseName;  
    //required setters and getters  
}

**Test.java**

package com.nareshit.client;  
  
import org.springframework.context.ApplicationContext;  
import org.springframework.context.support.ClassPathXmlApplicationContext;  
  
public class Test{  
public static void main(String args[]){  
      
    ApplicationContext context = new ClassPathXmlApplicationContext("com/nareshit/config/myBeans.xml");  
    Student stud = (Student)context.getBean("student");  
    stud.displayInfo();  
 }  
}

**myBeans.xml**

<?xml version="1.0" encoding="UTF-8"?>  
  
<beans xmlns="http://www.springframework.org/schema/beans"  
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
xmlns:p="http://www.springframework.org/schema/p"  
xsi:schemaLocation="http://www.springframework.org/schema/beans  
<http://www.springframework.org/schema/beans/spring-beans-3.0.xsd>">  
  
<bean id="student" class="com.nareshit.bean.Student">  
<property name="name" value="sathish"/>  
<property name="age" value="28"/>  
<property name="course" ref="course"/>  
</bean>  
  
<bean id="course" class=”com.nareshit.bean.Course">  
<property name="courseName" value="java"/>  
</bean>  
</beans>

In the above program dependencies of Student and Course class have been injected using <property> tags.  
  
Now the same program using p namespace is (you only need to change myBeans.xml, other classes remain same):  
  
**myBeans.xml**

<?xml version="1.0" encoding="UTF-8"?>  
  
<beans xmlns="http://www.springframework.org/schema/beans"  
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
xmlns:p="http://www.springframework.org/schema/p"  
xsi:schemaLocation="http://www.springframework.org/schema/beans  
<http://www.springframework.org/schema/beans/spring-beans-3.0.xsd>">  
  
<bean id="student" class=”com.nareshit.bean.Student" **p:name="sathish"** **p:age="28"** p:**course-ref=**"course"/>  
<bean id="course" class="com.nareshit.bean.Course” **p:courseName="java"**/>  
</beans>

In the above program we are using p namespace to inject the dependencies of Student and Course classes.  
  
For example,  
  
p:age="28" is substitute of:  
<property  name="age" value="28">

Note how we are injecting reference to other bean as a value of property -**course** of Student class, using p namespace:  
p:course-ref="course"   
is a substitute of   
<property name="course" ref="course">  
  
Note the **-ref** at the end of p:course-ref="course". **-ref** means this property would accept a reference to other bean as its value.  
  
Also note that unlike <property> tags, p namespace are not separate tags but they are the part of <bean> tag  
  
For p namepsace to work you have to add its schema URI <http://www.springframework.org/schema/p> in above xml file.

**C-Name Space :**

C-Namespace has been introduced in spring 3.1.1, in order to perform constructor injection we need to use<constructor-arg> tag.instead of writing the length <constructor-arg> tag,we can replace it with c:namespace.

Consider following example:  
  
**Student.java**

package com.nareshit.bean;  
  
public class Student {  
    private String name;  
    private int age;  
    private Course course;  
     
    //constructor  
    Student(String name, int age, Course course){  
        this.name = name;  
        this.age = age;  
        this.course = course;  
    }  
     
// setters and getters...     
  
    public void displayInfo(){  
        System.out.println("Student Name : "+name);  
        System.out.println("Student Age : "+ age);  
        System.out.println("Course name : "+course.getCourseName());  
    }  
}

**Course.java**

package com.nareshit.bean;  
  
public class Course {  
pivate    String courseName;  
    public  Course(String course){  
        this.courseName = course;  
    }  
      public String getCourseName(){  
        return courseName;  
         
    }  
     
}

**Test.java**

package com.nareshit.client;  
import org.springframework.context.ApplicationContext;  
import org.springframework.context.support.ClassPathXmlApplicationContext;  
  
public class Test{  
public static void main(String args[]){  
 ApplicationContext context = new ClassPathXmlApplicationContext("com/nareshit/config/myBeans.xml");  
    Student stud = (Student)context.getBean("student");  
    stud.displayInfo();  
 }  
}

**myBeans.xml**

<?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.xsd>">  
<bean id="student" class=" com.nareshit.bean.Student ">  
<constructor-arg value="sathish"/>  
<constructor-arg value="28"/>  
<constructor-arg ref="course"/>  
</bean>  
  
<bean id="course" class=”com.nareshi.bean.Course">  
<constructor-arg value="java"/>  
</bean>  
</beans>

In the above program arguments of Student and Course class constructors have been injected using <constructor-arg> tags.  
  
Now the same program using c namespace is (you only need to change myBeans.xml, other classes remain same):  
  
**myBeans.xml**

<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
xmlns:c="http://www.springframework.org/schema/c"  
xsi:schemaLocation="http://www.springframework.org/schema/beans  
<http://www.springframework.org/schema/beans/spring-beans.xsd>">  
  
<bean id="student" class="com.nareshit.bean.Student" c:name="sathish" c:age="28" c:course-ref="course"/>  
  
<bean id="course" class="com.nareshit.bean.Course" c:course="java"/>  
</beans>

In the above program we are using c namespace to inject the argument values of Student and Course class constructors. For example,  
  
c:age="28"  is substitute of:  
<constructor-arg value="28"/>  
  
Note how we are injecting reference to other bean as a value for the **'c'** argument of Student class constructor, using c namespace:  
  
c:course-ref="course"  
is a substitute of   
<constructor-arg ref="course"/>  
  
Note the **-ref** at the end of **c:course-ref="course".** **-ref** means this property would accept a reference to other bean as its value.  
  
Unlike <constructor-arg> tags, c namespace are not separate tags but they are the part of <bean> tag  
  
  
For c namespace to work you have to add its schema URI <http://www.springframework.org/schema/c> in above xml file.  
  
In case when we do not know the names of constructor arguments we can use following syntax in **myBeans.xml:**

<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
xmlns:c="http://www.springframework.org/schema/c"  
xsi:schemaLocation="http://www.springframework.org/schema/beans  
<http://www.springframework.org/schema/beans/spring-beans.xsd>">  
  
<bean id="student" class="com.nareshit.bean.Student" **c:\_0="abc" c:\_1="28" c:\_2-ref="course"**/>  
  
<bean id="course" class="com.nareshit.bean.Course" **c:\_0="java"**/>  
</beans>

Here c:\_0 means first argument of constructor, c:\_1 means second argument and c:\_2-ref means third argument which is accepting a reference to other bean as its value.