**Spring container**

The **Spring container** is at the core of the**Spring** Framework. The **container** will create the objects, wire them together, configure them, and manage their complete life cycle from creation till destruction. The**Spring container** uses DI to manage the components that make up an application.

It is similar how servlet container does .Servlet contain reads the web.xml and instantiate the servlets defined in the web.xml file.

Spring uses the BeanFactory which uses the Factrory design pattern to ceate spring beans from provided xml where it is defined how spring should create the objects and creates the object and provides it back.

BeanFactory factory=new XmlBeanFactory(new FileSystemResource(“Beans.xml”));

// ApplicationContex does everything that BeanFactory does and more than that.

ApplicationContex context=new ClassPathXmlApplicationContext(new FileSystemResource(“Beans.xml”));

* **BeanFactory:** Does not support the Annotation based dependency Injection.
* **ApplicationContext:** Support Annotation based dependency Injection. -@Autowired, @PreDestroy
* **BeanFactory:** Does not Support
* **ApplicationContext:** Application contexts can publish events to beans that are registered as listeners
* **BeanFactory:** Does not support way to access Message Bundle(internationalization (I18N)
* **ApplicationContext:** Support internationalization (I18N) messages.
* **BeanFactory:** Doesn’t support.
* **ApplicationContext:** Support many enterprise services such JNDI access, EJB integration, remoting.
* **BeanFactory:** By default its support Lazy loading
* **ApplicationContext:** It's By default support Aggresive loading.

|  |  |
| --- | --- |
|  | Spring provides two kinds of IOC container, one is XMLBeanFactory and other is ApplicationContext.  +---------------------------------------+-----------------+--------------------------------+  | | XMLBeanFactory | ApplicationContext |  +---------------------------------------+-----------------+--------------------------------+  | Annotation support | No | Yes |  | BeanPostProcessor Registration | Manual | Automatic |  | implimentation | XMLBeanFactory ClassPath/FileSystem/WebXmlApplicationContext|  | internationalization | No | Yes |  | Enterprise services | No | Yes |  | ApplicationEvent publication | No | Yes |  +---------------------------------------+-----------------+--------------------------------+  [enter image description here](https://i.stack.imgur.com/00grD.jpg)   1. FileSystemXmlApplicationContext: Beans loaded through the full path. 2. ClassPathXmlApplicationContext: Beans loaded through the CLASSPATH 3. WebXmlApplicationContext: Beans loaded through the web application context. |

**Injecting Bean Using BeanFactory**

BeanFactory factory=new XmlBeanFactory(new FileSystemResource(“Beans.xml”));

Triangle triangle =( Triangle)factory.getBean(“rectangle”);

On triangle object we can call the methods defined inside.

**Triangle Class**

class Triangle{

private String type;

//getters and setters

}

**Beans.xml**

<beans>

<bean id=”triangle” class=”com.xyz.Triangle”>

<property name=”type” value=”Equilateral Triangle” />

</bean>

</beans>

**Injecting Bean using ApplicationContex**

**Main Class**

ApplicationContex context=new ClassPathXmlApplicationContext(new FileSystemResource(“Beans.xml”));

Triangle triangle =(Triangle)context.getBean(“triangle”);

triangle.draw();

**Trangle Class**

Class Trangle{

priivate Point pontA;

priivate Point pontB;

priivate Point pontC;

//getters and setters

}

**Point Class**

class Point{

private int x;

private int y;

//getters and setters

}

// in this case Triangle bean is dependent on Point bean

**beans.xml**

<beans>

<bean id=”triangle” class=”com.xyz.Triangle”>

<property name=”pointA” ref=”point1” />

<property name=”pointB” ref=”point2” />

<property name=”pointC” ref=”point3” />

</bean>

<bean id=”point1” class=”com.xyz.Point”>

<property name=”x” value=”0”/>

<property name=”y” value=”0”/>

</bean>

<bean id=”point2” class=”com.xyz.Point”>

<property name=”x” value=”20”/>

<property name=”y” value=”30”/>

</bean>

<bean id=”point3” class=”com.xyz.Point”>

<property name=”x” value=”0”/>

<property name=”y” value=”40”/>

</bean>

</beans>

**Instanciating Bean using Constructor Way and Property way**

Class Trangle{

priivate int height;

priivate String type;

public Triangle(int height,int type){

this.height=height;

this.type=type;

}

//getters and setters

}

**beans.xml**

<beans>

<bean id=”triangle” class=”com.xyz.Triangle”>

<property name=”height” ref=”10” />

<property name=”type” ref=”Equilateral” />

</bean>

</beans>

In the above approach we are instanciating the class Trangle using property’s setter method.

**Constructor approch**

Class Trangle{

priivate int height;

priivate String type;

//getters and setters

}

<beans>

<bean id=”triangle” class=”com.xyz.Triangle”>

<construtor-arg type =” int” name=”height” ref=”10” />

<construtor-arg type=”lava.lang.String” name=”type” ref=”Equilateral” />

</bean>

</beans>

//for overloaded constructor we have to give type of argument

Another approch is index number

<beans>

<bean id=”triangle” class=”com.xyz.Triangle”>

<construtor-arg index =” 0” name=”height” ref=”10” />

<construtor-arg inex=”1” name=”type” ref=”Equilateral” />

</bean>

</beans>

**InnerBean**

Inner is used in those cases if the bean is going to be used only once and should be part of one parent bean.

In that case the inner bean does not have to have a id and the the parent property should not have a ref .

**beans.xml**

<beans>

<bean id=”triangle” class=”com.xyz.Triangle” name=”triangle-name”>

<property name=”pointA” >

<bean class=”com.xyz.Point”>

<property name=”x” value=”0”/>

<property name=”y” value=”0”/>

</bean>

</property>

<property name=”pointB” >

<bean class=”com.xyz.Point”>

<property name=”x” value=”20”/>

<property name=”y” value=”30”/>

</bean>

</property>

<property name=”pointC” ref=”point3” />

</bean>

<bean id=”point3” class=”com.xyz.Point”>

<property name=”x” value=”0”/>

<property name=”y” value=”40”/>

</bean>

<alias name=”triangle” alias=”triangle-alias” />

</beans>

BeanFactory factory=new XmlBeanFactory(new FileSystemResource(“Beans.xml”));

Triangle triangle =(Triangle)context.getBean(“triangle”);

Triangle triangle =(Triangle)context.getBean(“triangle-alias”);

Triangle triangle =(Triangle)context.getBean(“triangle-name”);

//both are same

**beans.xml**

<beans>

<bean id=”triangle” class=”com.xyz.Triangle” name=”triangle-name”>

<property name=”pointA” >

<bean class=”com.xyz.Point”>

<property name=”x” value=”0”/>

<property name=”y” value=”0”/>

</bean>

</property>

<property name=”pointB” >

<bean class=”com.xyz.Point”>

<property name=”x” value=”20”/>

<property name=”y” value=”30”/>

</bean>

</property>

<property name=”pointC” >

<idref= “point3”/> // this line will check any bean is there with id point3 in the xml

</property>

</bean>

<bean id=”point3” class=”com.xyz.Point”>

<property name=”x” value=”0”/>

<property name=”y” value=”40”/>

</bean>

<alias name=”triangle” alias=”triangle-alias” />

</beans>

**Collection of Bean**

Class Trangle{

priivate List<Point> points;

//getters and setters

}

**Point Class**

class Point{

private int x;

private int y;

//getters and setters

}

// in this case Triangle bean is dependent on Point bean

**beans.xml**

<beans>

<bean id=”triangle” class=”com.xyz.Triangle”>

<property name=”points” >

<list>

<ref bean=” point1” />

<ref bean=” point2” />

<ref bean=” point3” />

</list>

</property>

</bean>

<bean id=”point1” class=”com.xyz.Point”>

<property name=”x” value=”0”/>

<property name=”y” value=”0”/>

</bean>

<bean id=”point2” class=”com.xyz.Point”>

<property name=”x” value=”20”/>

<property name=”y” value=”30”/>

</bean>

<bean id=”point3” class=”com.xyz.Point”>

<property name=”x” value=”0”/>

<property name=”y” value=”40”/>

</bean>

</beans>

// for set and map we can use similarly

**Customer Class**

public class Customer

{

private List<Object> lists;

private Set<Object> sets;

private Map<Object, Object> maps;

private Properties pros;

//...

}

**Beans.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="CustomerBean" class="com.mkyong.common.Customer">

<!-- java.util.List -->

<property name="lists">

<list>

<value>1</value>

<ref bean="PersonBean" />

<bean class="com.mkyong.common.Person">

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

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

<property name="age" value="28" />

</bean>

</list>

</property>

<!-- java.util.Set -->

<property name="sets">

<set>

<value>1</value>

<ref bean="PersonBean" />

<bean class="com.mkyong.common.Person">

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

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

<property name="age" value="28" />

</bean>

</set>

</property>

<!-- java.util.Map -->

<property name="maps">

<map>

<entry key="Key 1" value="1" />

<entry key="Key 2" value-ref="PersonBean" />

<entry key="Key 3">

<bean class="com.mkyong.common.Person">

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

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

<property name="age" value="28" />

</bean>

</entry>

</map>

</property>

<!-- java.util.Properties -->

<property name="pros">

<props>

<prop key="admin">admin@nospam.com</prop>

<prop key="support">support@nospam.com</prop>

</props>

</property>

</bean>

<bean id="PersonBean" class="com.mkyong.common.Person">

<property name="name" value="mkyong1" />

<property name="address" value="address 1" />

<property name="age" value="28" />

</bean>

</beans>

**App Class**

public class App

{

public static void main( String[] args )

{

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

Customer cust = (Customer)context.getBean("CustomerBean");

System.out.println(cust);

}

}

**Autowiring Of Beans**

If we apply autowire attribute to byName in triangle bean then spring will search the bean.xml where any bean is declared with id same as property of Triangle class property name.

Not a good practice.

Other possible values

byName

byType 🡪 one type per bean (two property of same type not allowed)

constructor 🡪 one type per bean (two property of same type not allowed)

**Triangle Class**

Class Trangle{

priivate Point pontA;

priivate Point pontB;

priivate Point pontC;

//getters and setters

}

**Point Class**

class Point{

private int x;

private int y;

//getters and setters

}

**Beans.xml**

<beans>

<bean id=”triangle” class=”com.xyz.Triangle” autowire=”byName”>

</bean>

<bean id=”pointA” class=”com.xyz.Point”>

<property name=”x” value=”0”/>

<property name=”y” value=”0”/>

</bean>

<bean id=”pointB” class=”com.xyz.Point”>

<property name=”x” value=”20”/>

<property name=”y” value=”30”/>

</bean>

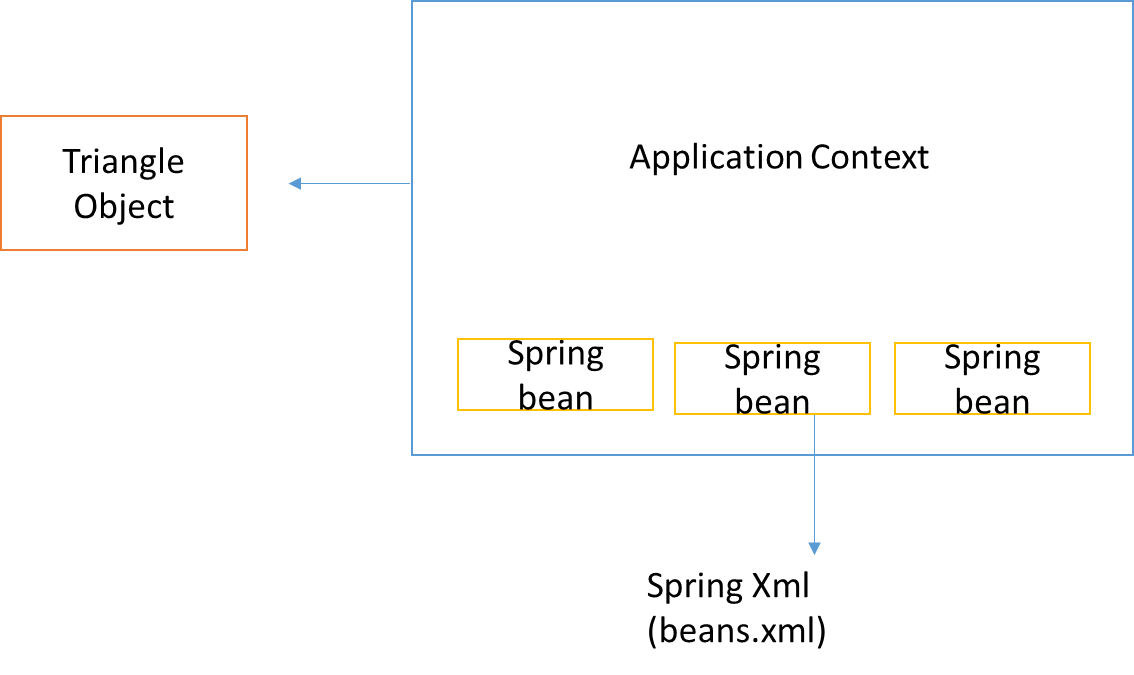
<bean id=”pointC” class=”com.xyz.Point”>

<property name=”x” value=”0”/>

<property name=”y” value=”40”/>

</bean>

</beans>



As soon as application context object is initialized the application context reads the file and creates the beans even before getBean method is called. This is default behavior of application context.

That’s why The default behavior of every bean is singleton.ie even if we call getBean multiple times it will return the same object that got created during initialization of application context.

It is possible multiple context running on same jvm in that case singleton means per context one object will created.

Beans can be of two types

1. Singleton 🡪 only once created
2. Prototype 🡪 every time getBean creates a new bean
3. Request 🡪 New Bean per servlet request
4. Session 🡪 New Bean per session.
5. Global Session 🡪New bean per global HTTP session (portlet context)

<beans>

<bean id=”triangle” class=”com.xyz.Triangle” scope=”singleton/prototype/request/session”>

-----------------------------

</bean>

</beans>

**Spring bean Definition Inheritance**

triangle1 and triangle2 inherit the pointA from the parent triangle

abstract=”true” will stop application creating bean for parenttriangle

**beans.xml**

<beans>

<bean id=”parenttriangle” class=”com.xyz.Triangle” abstract=”true”>

<property name=”pointA” ref=”point1” />

</bean>

<bean id=”triangle1” class=”com.xyz.Triangle” parent=”parenttriangle”>

<property name=”pointB” ref=”point2” />

<property name=”pointC” ref=”point3” />

</bean>

<bean id=”triangle2” class=”com.xyz.Triangle” parent=”parenttriangle”>

<property name=”pointB” ref=”point2” />

<property name=”pointC” ref=”point3” />

</bean>

<bean id=”point1” class=”com.xyz.Point”>

<property name=”x” value=”0”/>

<property name=”y” value=”0”/>

</bean>

<bean id=”point2” class=”com.xyz.Point”>

<property name=”x” value=”20”/>

<property name=”y” value=”30”/>

</bean>

<bean id=”point3” class=”com.xyz.Point”>

<property name=”x” value=”0”/>

<property name=”y” value=”40”/>

</bean>

</beans>

We can inherit the collection, we can override a collection defined in parent bean we can merge also.

**beans.xml**

<beans>

<bean id=”parenttriangle” class=”com.xyz.Triangle” abstract=”true”>

<property name=”points” > // points is a list type property in triangle bean

<list>

<ref bean=”point1”/>

</list>

</property>

</bean>

<bean id=”triangle1” class=”com.xyz.Triangle” parent=”parenttriangle”>

<property name=”points” >

<list merge=”true”> // if merge is not true child bean will override the parent points

<ref bean=”point1”/>

</list>

</property>

</bean>

<bean id=”point1” class=”com.xyz.Point”>

<property name=”x” value=”0”/>

<property name=”y” value=”0”/>

</bean>

<bean id=”point2” class=”com.xyz.Point”>

<property name=”x” value=”20”/>

<property name=”y” value=”30”/>

</bean>

<bean id=”point3” class=”com.xyz.Point”>

<property name=”x” value=”0”/>

<property name=”y” value=”40”/>

</bean>

</beans>

**ApplicationContextAware**

By implementing ApplicationContextAware we have implement below method in our bean

@override

public void setApplicationContext(ApplicationContext context) throws BeansException

{

this.context = context;

}

Than mean we have context in our bean .So in case where triangle object is of type singleton mentioned in beans.xml file but pointA, pointB, pointC are declared as prototype. When triangle object will be created application context will return the same object again and again even the inner objects points will the same even if those are declared as prototype.

To created new pointA, pointB, pointC objects every time triangle bean is created we need the context inside the triangle object .we can read the bean. Xml file to create pointA, pointB, pointC.

**BeanNameAware**

If Bean class implements this interface we can get the bean name defined in the beans.xml file.

@override

Public void setBeanName (String name)

{

}

**Close Application Context**

If we are writing desktop application we need to close the application context.

We have to t

Public static void main{

AbstractApplicationContex context=new ClassPathXmlApplicationContext(new FileSystemResource(“Beans.xml”));

context.registerShutdownHook();

Triangle triangle =( Triangle)factory.getBean(“rectangle”);

Triangle.draw();

}

To know once object is initialized and when object is destroyed below interfaces are used

InitializedBean 🡪 afterPropertiesSet() method we need to override in bean

DisposableBean 🡪 destroy() method we need to override in bean

Without implementing InitializedBean interface we can call on user defined init method

By mentioning the below property in the bean’s declaration in xml and putting the mentioned method in the bean. Similarly for DisposableBean.

<bean id=”triangle” init-method=”myInit” destroy-method=”myDestroy”>

</bean>

To add init and destroy method at global level we need to use below property in <beans> tag of xml.These methods will be called for if it exists in the bean otherwise will be ignored.

<beans default-init-method=” myInit” default-destroy-method=” myDestroy”>

</beans>

**Bean post processors**

BeanPostProcessor is used to interact with newly created bean instances *before* and/or *after* their initialization method is invoked by the Spring container. You can use BeanPostProcessor to execute custom logic *before* and/or *after* bean’s initialization method is invoked by the Spring container.

BeanPostProcessor interface defines the following methods:

* Object postProcessBeforeInitialization(Object bean, String beanName) – this method is invoked *before* the initialization method of a bean instance is invoked
* Object postProcessAfterInitialization(Object bean, String beanName) – this method is invoked *after* the initialization method of a bean instance is invoked

BeanPostProcessor’s methods accept newly created bean instance and its name as arguments, and return the same or modified bean instance. You configure a BeanPostProcessor implementation in the application context XML file like any other Spring bean. Once the BeanPostProcessor beans are created, the Spring container invokes each BeanPostProcessor’s postProcessBeforeInitialization and postProcessAfterInitialization methods for each bean instance created by the Spring container.

### BeanPostProcessor example – Validating bean instances

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

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

**Example listing**– InstanceValidator interface

package sample.spring.chapter04.springbankapp.common;

public interface InstanceValidator {

  void validateInstance();

}

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

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

**Example listing** – FixedDepositDaoImpl class

package sample.spring.chapter04.springbankapp.dao;

import org.apache.log4j.Logger;

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

public class FixedDepositDaoImpl implements FixedDepositDao, InstanceValidator {

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

  private DatabaseConnection connection;

  public FixedDepositDaoImpl() {

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

  }

  public void initializeDbConnection() {

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

   connection = DatabaseConnection.getInstance();

  }

  @Override

  public void validateInstance() {

    logger.info("Validating FixedDepositDaoImpl instance");

     if(connection == null) {

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

   }

  }

}

In the above example listing, the initializeDbConnection method is the initialization method that retrieves an instance of DatabaseConnection by calling getInstance *static* method of DatabaseConnection class. The connection attribute is null if FixedDepositDaoImpl instance fails to retrieve an instance of DatabaseConnection. If connection attribute is null, the validateInstance method logs an error message indicating that the FixedDepositDaoImpl instance is not correctly initialized. As the initializeDbConnection initialization method sets the value of connection attribute, the validateInstance method *must* be invoked *after* the initializeDbConnection method. In a real world application development scenario, if a bean instance is not configured correctly, the validateInstance method may take some corrective action or throw a runtime exception to stop the application from starting up. For simplicity, the validateInstance method logs an error message if a bean instance is not configured correctly.

The following example listing shows the InstanceValidationBeanPostProcessor class that implements Spring’s BeanPostProcessor interface, and is responsible for invoking validateInstance method of newly created beans:

**Example listing**– InstanceValidationBeanPostProcessor class

package sample.spring.chapter04.springbankapp.postprocessor;

import org.springframework.beans.BeansException;

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

import org.springframework.core.Ordered;

public class InstanceValidationBeanPostProcessor implements BeanPostProcessor, Ordered {

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

    private int order;

    public InstanceValidationBeanPostProcessor() {

        logger.info("Created InstanceValidationBeanPostProcessor instance");

    }

    @Override

    public Object postProcessBeforeInitialization(Object bean, String beanName)

            throws BeansException {

        logger.info("postProcessBeforeInitialization method invoked");

        return bean;

    }

    @Override

    public Object postProcessAfterInitialization(Object bean, String beanName)

            throws BeansException {

        logger.info("postProcessAfterInitialization method invoked");

        if (bean instanceof InstanceValidator) {

            ((InstanceValidator) bean).validateInstance();

        }

        return bean;

    }

    public void setOrder(int order) {

        this.order = order;

    }

    @Override

    public int getOrder() {

        return order;

    }

}

The above example listing shows that the InstanceValidationBeanPostProcessor class implements Spring’s BeanPostProcessor and Ordered interfaces. The postProcessBeforeInitialization method simply returns the bean instance passed to the method. In the postProcessAfterInitialization method, if the bean instance is found to be of type InstanceValidator, the bean instance’s validateInstance method is invoked. This means that if a bean implements InstanceValidator interface, InstanceValidationBeanPostProcessor calls validateInstance method of the bean instance *after* the initialization method of the bean instance is invoked by the Spring container.

The Ordered interface defines a getOrder method which returns an integer value. The integer value returned by the getOrder method determines the priority of a BeanPostProcessor implementation with respect to other BeanPostProcessor implementations configured in the application context XML file. A BeanPostProcessor with *higher* order value is considered at a *lower* priority, and is executed *after* the BeanPostProcessor implementations with *lower* order values are executed.

The following example listing shows bean definitions for InstanceValidationBeanPostProcessor class:

**Example listing**– InstanceValidationBeanPostProcessor bean definition

<bean class="…...springbankapp.postprocessor.InstanceValidationBeanPostProcessor">

  <property name="order" value="1" />

</bean>

In the above bean definition, <bean> element’s id attribute is *not* specified because we typically don’t want InstanceValidationBeanPostProcessor to be a dependency of any other bean. The <property> element sets the value of order property to 1.

**BeanFactoryPostProcessor**

* [PropertyOverrideConfigurer](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/beans/factory/config/PropertyOverrideConfigurer.html) for "beanName.property=value" style overriding (*pushing* values from a properties file into bean definitions)
* [PropertyPlaceholderConfigurer](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/beans/factory/config/PropertyPlaceholderConfigurer.html) for replacing "${...}" placeholders (*pulling* values from a properties file into bean definitions)