**MVC application: Struts**

**Spring framework:**

Spring has various modules which can be used as per the requirement

1. Spring core: desktop application
2. Spring MVC: web application
3. Spring data jpa: integrating with jpa framework
4. Spring jdbc: integrating with jdbc
5. Spring data rest: restful webservice
6. Spring security
7. Spring transactions
8. Spring boot: ease ur development
9. Spring AOP: separating ur business logic from service layer
10. Spring Cloud

Light-weighted framework. So, you can include a particular module in ur application as per ur reqt. Spring is very simple.

**Spring core: desktop application**

Spring is used mostly because of a feature which is called **Dependency Injection.** It is also called Inversion of Control. (IOC). It provides u loose coupling between classes.

Spring provides you a container which can create objects of your bean classes and do the dependency injection.

Class A{

B b= new BImpl();

}

//If tomorrow I need to change B to some other impl class, I would have to do changes in A class also

Interface B{}

Class BImpl implements B{

//outdated

}

Class BNewImpl implements B{

}

To prevent it, to provide loose coupling: we use spring

If I do changes in one class and it affects other classes: Bad design (Tight coupling)

Class A{

B b;//interface

}

**Spring Config:**

1. Xml based config
2. Java based config
3. Annotation based config

Spring has provided a container in the form of interfaces. So when u create the object of the impl classes for these interface, it will start your spring container.

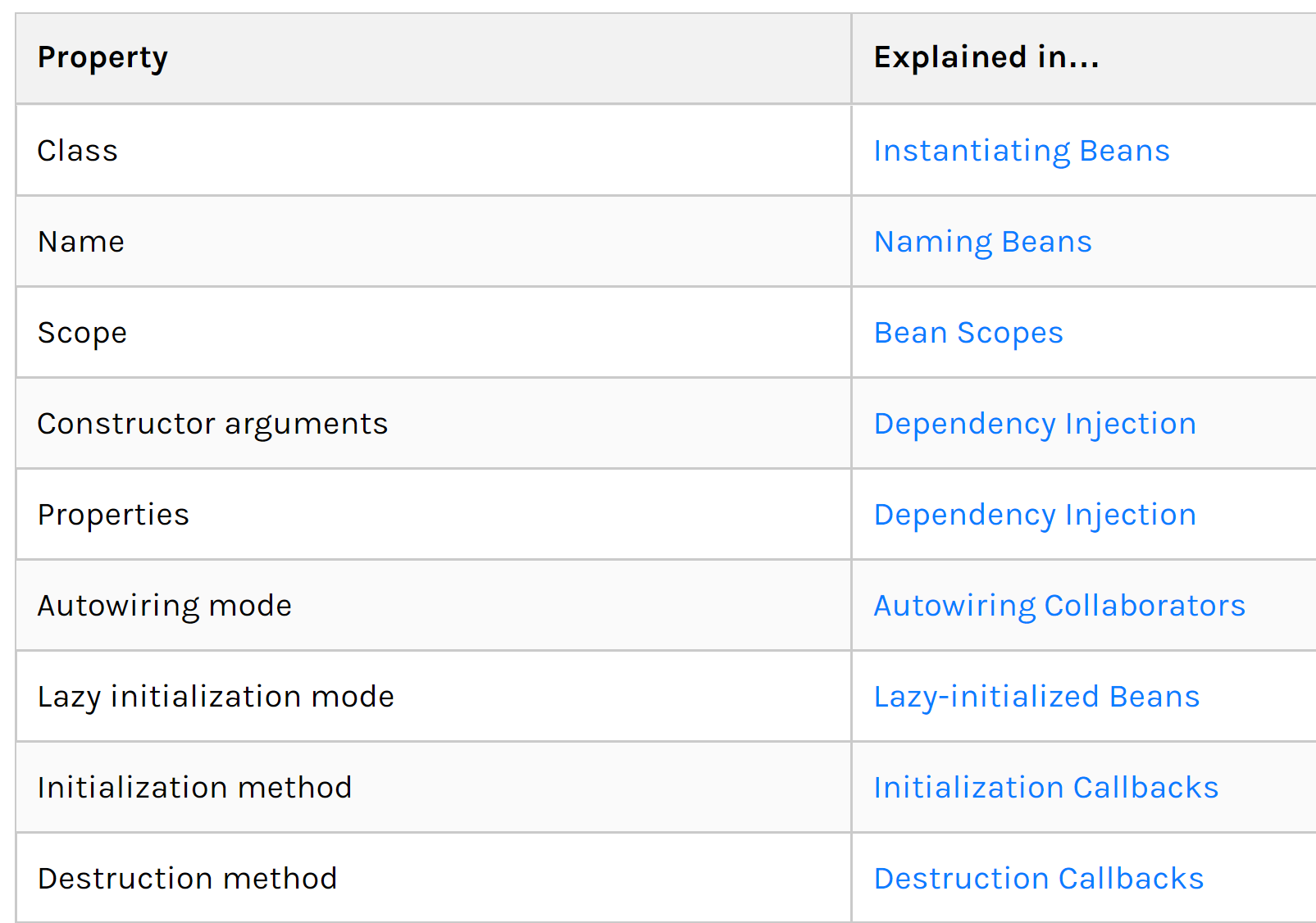
1. BeanFactory: simple container.. simple application
2. ApplicationContext: advanced container, lot more features.

XSD/ DTD: xml validation (valid tags)

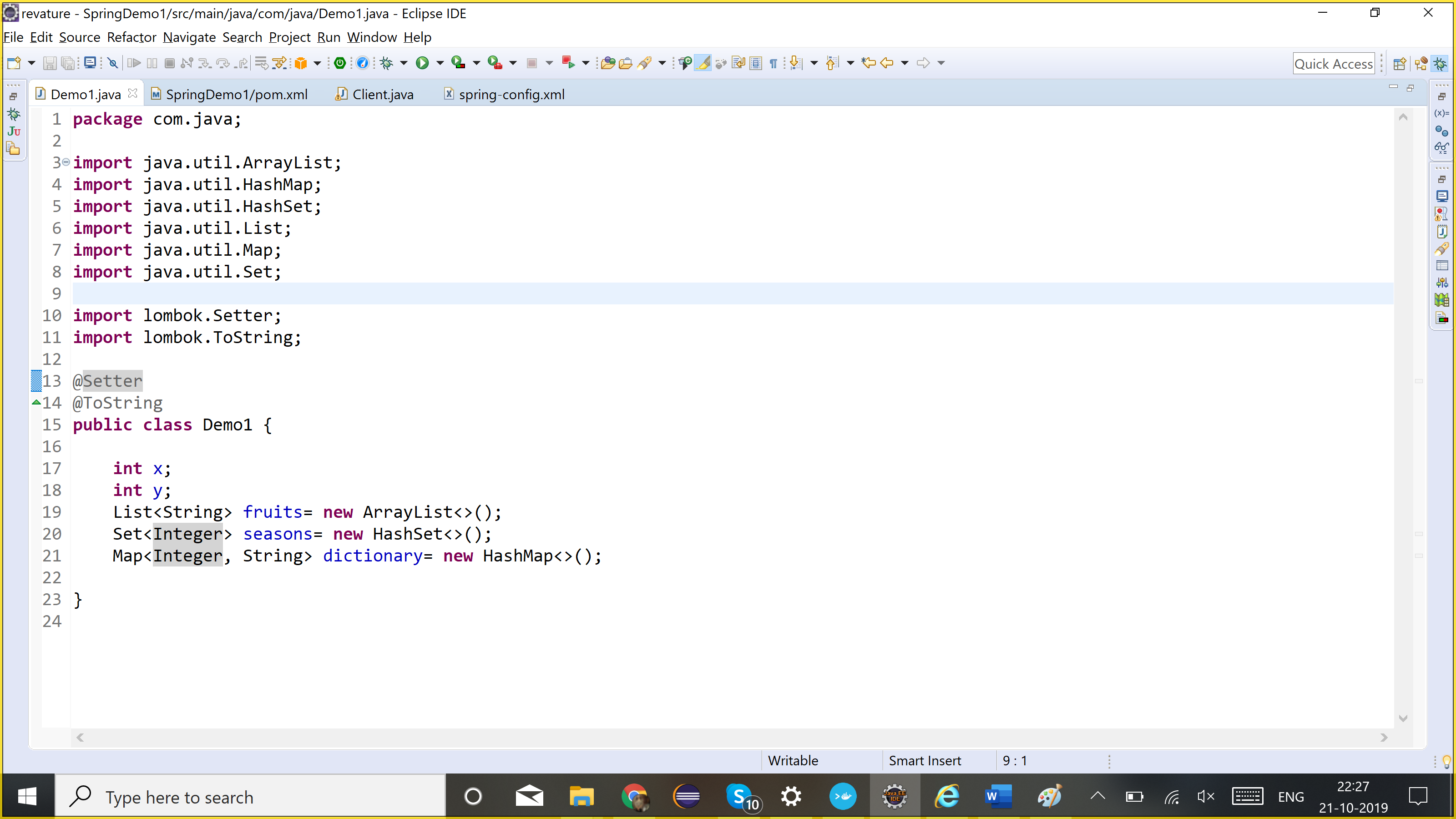
Constructor Injection: When we are using a constructor to set the dependency instead of using a setter

Using a setter: <property>

Constructor: <constructor-arg>



Initializing bean:



Every bean by-default is of a singleton scope

Demo1 obj1= container.getBean(Demo1.**class**);

Demo1 obj2= container.getBean(Demo1.**class**);

System.***out***.println(obj1==obj2);//same object : true:



getBean(), it should return me a different object:



<?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 class=*"com.java.Demo1"* id=*"d1"*>

<property name=*"x"* value=*"5"*></property>

<property name=*"y"* value=*"10"*></property>

<property name=*"fruits"*>

<list>

<value>Apple</value>

<value>Guava</value>

</list>

</property>

<property name=*"seasons"*>

<set>

<value>12</value>

<value>23</value>

</set>

</property>

<property name=*"dictionary"*>

<map>

<entry key=*"1"* value=*"Abacus"*/>

<entry key=*"2"* value=*"Abate"*></entry>

</map>

</property>

</bean>

<bean class=*"com.java.Demo1"* id=*"d2"* scope=*"prototype"*>

<property name=*"x"* value=*"1"*></property>

<property name=*"y"* value=*"20"*></property>

<property name=*"fruits"*>

<list>

<value>Apple</value>

<value>Guava</value>

</list>

</property>

<property name=*"seasons"*>

<set>

<value>12</value>

<value>23</value>

</set>

</property>

<property name=*"dictionary"*>

<map>

<entry key=*"1"* value=*"Abacus"*/>

<entry key=*"2"* value=*"Abate"*></entry>

</map>

</property>

</bean>

</beans>

**public** **class** Client {

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

BeanFactory container= **new** ~~XmlBeanFactory~~(**new** ClassPathResource("spring-config.xml"));//start my spring container

//lazy initialization: getBean: create the object of that class

//ApplicationContext container= new ClassPathXmlApplicationContext("spring-config.xml");

//eager initialization: read config file and create objects of all beans defined in config file

Demo1 obj1= container.getBean("d1", Demo1.**class**);//it will only create object what class u ask for

Demo1 obj2= container.getBean("d1", Demo1.**class**);//whether it is a same object or diff object??

//Every bean by-default is of a singleton scope

System.***out***.println(obj1==obj2);//same object : true:

Demo1 obj3= container.getBean("d2", Demo1.**class**);

System.***out***.println(obj1==obj3);//false

}

}

**Autowiring: automatically will inject the dependency**

Bydefault ayutowire= “no”

1. byType: If it finds any bean of that type, it will automatically inject the dependency . It works for setter injection. But must have only one bean defined for that type
2. byName: When u have more than 1 bean defined, so it will inject bean with the same name.
3. constructor: for constructor injection, bydefault does it by type, but if 2 beans of same type, then will try to do it by name.
4. no: default

**package** com.java;

**import** lombok.Setter;

**public** **class** Demo1 {

Demo2 obj1;

Demo1(Demo2 obj1){

**this**.obj1= obj1;

}

}

@Setter

**class** Demo2{

Demo3 obj;

}

**class** Demo3{

**int** x;

}

<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 class=*"com.java.Demo3"* id=*"obj"*>

<property name=*"x"* value=*"3"*></property>

</bean>

<bean class=*"com.java.Demo3"* id=*"d3b"*>

<property name=*"x"* value=*"3"*></property>

</bean>

<bean class=*"com.java.Demo2"* id=*"obj1"* autowire=*"byName"*>

<!-- <property name="obj" ref="d3" /> -->

</bean>

<bean class=*"com.java.Demo2"* id=*"d2b"* autowire=*"byName"*>

<!-- <property name="obj" ref="d3" /> -->

</bean>

<bean class=*"com.java.Demo1"* id=*"d1"* autowire=*"constructor"*>

<!-- <constructor-arg ref="d2" /> -->

</bean>

</beans>

**Constructor type injection, type:**

**public** **class** **ExampleBean** {

*// Number of years to calculate the Ultimate Answer*

**private** **int** years;

*// The Answer to Life, the Universe, and Everything*

**private** String ultimateAnswer;

**public** **ExampleBean**(**int** years, String ultimateAnswer) {

**this**.years = years;

**this**.ultimateAnswer = ultimateAnswer;

}

}

<bean id="exampleBean" class="examples.ExampleBean">

<constructor-arg type="int" value="7500000"/>

<constructor-arg type="java.lang.String" value="42"/>

</bean>

**Using index:**

<bean id="exampleBean" class="examples.ExampleBean">

<constructor-arg index="0" value="7500000"/>

<constructor-arg index="1" value="42"/>

</bean>

**Using name:**

<bean id="exampleBean" class="examples.ExampleBean">

<constructor-arg name="years" value="7500000"/>

<constructor-arg name="ultimateAnswer" value="42"/>

</bean>

**Destroy-method**

<bean id="myDataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">

*<!-- results in a setDriverClassName(String) call -->*

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

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

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

<property name="password" value="masterkaoli"/>

</bean>

**Default init and destroy method:**



**Inner Beans:**

<bean id="outer" class="...">

*<!-- instead of using a reference to a target bean, simply define the target bean inline -->*

<property name="target">

<bean class="com.example.Person"> *<!-- this is the inner bean -->*

<property name="name" value="Fiona Apple"/>

<property name="age" value="25"/>

</bean>

</property>

</bean>

An inner bean definition does not require a defined ID or name

 If specified, the container does not use such a value as an identifier. The container also ignores the scope flag on creation, because inner beans are always anonymous and are always created with the outer bean. It is not possible to access inner beans independently or to inject them into collaborating beans other than into the enclosing bean.

**Set null value:**

<bean class="ExampleBean">

<property name="email">

<null/>

</property>

</bean>

**Aware**

**Using depends-on**

If a bean is a dependency of another bean, that usually means that one bean is set as a property of another. Typically you accomplish this with the [<ref/> element](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-ref-element) in XML-based configuration metadata. However, sometimes dependencies between beans are less direct. An example is when a static initializer in a class needs to be triggered, such as for database driver registration. The depends-on attribute can explicitly force one or more beans to be initialized before the bean using this element is initialized. The following example uses the depends-on attribute to express a dependency on a single bean:

<bean id="beanOne" class="ExampleBean" depends-on="manager"/>

<bean id="manager" class="ManagerBean" />

#### Lazy-initialized Beans

By default, ApplicationContext implementations eagerly create and configure all [singleton](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-scopes-singleton) beans as part of the initialization process. A lazy-initialized bean tells the IoC container to create a bean instance when it is first requested, rather than at startup.

In XML, this behavior is controlled by the lazy-init attribute on the <bean/> element, as the following example shows:

<bean id="lazy" class="com.something.ExpensiveToCreateBean" lazy-init="true"/>

<bean name="not.lazy" class="com.something.AnotherBean"/>

However, when a lazy-initialized bean is a dependency of a singleton bean that is not lazy-initialized, the ApplicationContext creates the lazy-initialized bean at startup, because it must satisfy the singleton’s dependencies. The lazy-initialized bean is injected into a singleton bean elsewhere that is not lazy-initialized

You can also control lazy-initialization at the container level by using the default-lazy-init attribute on the <beans/> element, a the following example shows:

<beans default-lazy-init="true">

*<!-- no beans will be pre-instantiated... -->*

</beans>

Avoid autowiring for a bean definition by setting its **autowire-candidate** attributes to false

Designate a single bean definition as the primary candidate by setting the **primary** attribute of its <bean/> element to true.

##### **Lookup Method Injection**

Lookup method injection is the ability of the container to override methods on container-managed beans and return the lookup result for another named bean in the container.  The Spring Framework implements this method injection by using bytecode generation from the CGLIB library to dynamically generate a subclass that overrides the method. For this dynamic subclassing to work, the class that the Spring bean container subclasses cannot be final, and the method to be overridden cannot be final, either.

**public** **abstract** **class** **CommandManager** {

**public** Object **process**(Object commandState) {

*// grab a new instance of the appropriate Command interface*

Command command = createCommand();

*// set the state on the (hopefully brand new) Command instance*

command.setState(commandState);

**return** command.execute();

}

*// okay... but where is the implementation of this method?*

**protected** **abstract** Command **createCommand**();

}

If the method is abstract, the dynamically-generated subclass implements the method. Otherwise, the dynamically-generated subclass overrides the concrete method defined in the original class. Consider the following example:

*<!-- a stateful bean deployed as a prototype (non-singleton) -->*

<bean id="myCommand" class="fiona.apple.AsyncCommand" scope="prototype">

*<!-- inject dependencies here as required -->*

</bean>

*<!-- commandProcessor uses statefulCommandHelper -->*

<bean id="commandManager" class="fiona.apple.CommandManager">

<lookup-method name="createCommand" bean="myCommand"/>

</bean>

Alternatively, within the annotation-based component model, you can declare a lookup method through the @Lookup annotation, as the following example shows:

**public** **abstract** **class** **CommandManager** {

**public** Object **process**(Object commandState) {

Command command = createCommand();

command.setState(commandState);

**return** command.execute();

}

**@Lookup**("myCommand")

**protected** **abstract** Command **createCommand**();

}

If you use Spring’s IoC container in a non-web application environment (for example, in a rich client desktop environment), register a shutdown hook with the JVM. Doing so ensures a graceful shutdown and calls the relevant destroy methods on your singleton beans so that all resources are released. You must still configure and implement these destroy callbacks correctly.

To register a shutdown hook, call the **registerShutdownHook**() method that is declared on the ConfigurableApplicationContext interface, as the following example shows:

**public** **final** **class** **Boot** {

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

ConfigurableApplicationContext ctx = **new** ClassPathXmlApplicationContext("beans.xml");

*// add a shutdown hook for the above context...*

ctx.registerShutdownHook();

*// app runs here...*

*// main method exits, hook is called prior to the app shutting down...*

}

}

Besides ApplicationContextAware and BeanNameAware (discussed [earlier](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-aware)), Spring offers a wide range of Aware callback interfaces that let beans indicate to the container that they require a certain infrastructure dependency. As a general rule, the name indicates the dependency type. The following table summarizes the most important Aware interfaces:

| **Table 4. Aware interfaces** | | |
| --- | --- | --- |
| **Name** | **Injected Dependency** | **Explained in…​** |
| ApplicationContextAware | Declaring ApplicationContext. | [ApplicationContextAware and BeanNameAware](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-aware) |
| ApplicationEventPublisherAware | Event publisher of the enclosing ApplicationContext. | [Additional Capabilities of the ApplicationContext](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#context-introduction) |
| BeanClassLoaderAware | Class loader used to load the bean classes. | [Instantiating Beans](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-class) |
| BeanFactoryAware | Declaring BeanFactory. | [ApplicationContextAware and BeanNameAware](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-aware) |
| BeanNameAware | Name of the declaring bean. | [ApplicationContextAware and BeanNameAware](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-aware) |
| BootstrapContextAware | Resource adapter BootstrapContext the container runs in. Typically available only in JCA-aware ApplicationContext instances. | [JCA CCI](https://docs.spring.io/spring/docs/current/spring-framework-reference/integration.html#cci) |
| LoadTimeWeaverAware | Defined weaver for processing class definition at load time. | [Load-time Weaving with AspectJ in the Spring Framework](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#aop-aj-ltw) |
| MessageSourceAware | Configured strategy for resolving messages (with support for parametrization and internationalization). | [Additional Capabilities of the ApplicationContext](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#context-introduction) |
| NotificationPublisherAware | Spring JMX notification publisher. | [Notifications](https://docs.spring.io/spring/docs/current/spring-framework-reference/integration.html#jmx-notifications) |
| ResourceLoaderAware | Configured loader for low-level access to resources. | [Resources](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#resources) |
| ServletConfigAware | Current ServletConfig the container runs in. Valid only in a web-aware Spring ApplicationContext. | [Spring MVC](https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html#mvc) |
| ServletContextAware | Current ServletContext the container runs in. Valid only in a web-aware Spring ApplicationContext. | [Spring MVC](https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html#mvc) |

Note again that using these interfaces ties your code to the Spring API and does not follow the Inversion of Control style. As a result, we recommend them for infrastructure beans that require programmatic access to the container.

If you want to implement some custom logic after the Spring container finishes instantiating, configuring, and initializing a bean, you can plug in one or more custom **BeanPostProcessor** implementations.

You can configure multiple BeanPostProcessor instances, and you can control the order in which these BeanPostProcessor instances execute by setting the **order** property. You can set this property only if the BeanPostProcessor implements the **Ordered** interface. If you write your own BeanPostProcessor, you should consider implementing the Ordered interface, too.

##### **PropertySourcesPlaceholderConfigurer**

You can use the PropertySourcesPlaceholderConfigurer to externalize property values from a bean definition in a separate file by using the standard Java Properties format. Doing so enables the person deploying an application to customize environment-specific properties, such as database URLs and passwords, without the complexity or risk of modifying the main XML definition file or files for the container.

<bean class="org.springframework.context.support.PropertySourcesPlaceholderConfigurer">

<property name="locations" value="classpath:com/something/jdbc.properties"/>

</bean>

<bean id="dataSource" destroy-method="close"

class="org.apache.commons.dbcp.BasicDataSource">

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

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

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

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

</bean>

jdbc.driverClassName=org.hsqldb.jdbcDriver

jdbc.url=jdbc:hsqldb:hsql://production:9002

jdbc.username=sa

jdbc.password=root

With the context namespace introduced in Spring 2.5, you can configure property placeholders with a dedicated configuration element. You can provide one or more locations as a comma-separated list in the location attribute, as the following example shows:

<context:property-placeholder location="classpath:com/something/jdbc.properties"/>

By default, if it cannot find a property in the specified properties files, it checks against Spring Environment properties and regular Java System properties.

**System properties** are set on the Java command line using the -Dpropertyname=value syntax

**@Autowired** annotation on such a constructor is no longer necessary if the target bean defines only one constructor to begin with. However, if several constructors are available, at least one must be annotated with @Autowired in order to instruct the container which one to use.

#### Injection with @Resource

Spring also supports injection by using the JSR-250 @Resource annotation (javax.annotation.Resource) on fields or bean property setter methods.

@Resource takes a name attribute. By default, Spring interprets that value as the bean name to be injected

**public** **class** **SimpleMovieLister** {

**private** MovieFinder movieFinder;

**@Resource**(name="myMovieFinder")

**public** **void** **setMovieFinder**(MovieFinder movieFinder) {

**this**.movieFinder = movieFinder;

}

}

If no name is explicitly specified, the default name is derived from the field name or setter method. In case of a field, it takes the field name. In case of a setter method, it takes the bean property name

**Static factory method**

**Factory-method**

**Init methods**

**Bean processors**

**Depends-on**

**Java config: @Bean, @Import, @Lazy, @DependsOn**

**Annotation based config: @Required, @Autowired, @PostConstruct, @PreDestory, @Value,**

**Static factory method vs instance factory method**

**package** com.java;

**public** **class** A {

**static** A *obj*;

//<bean class="com.java.A" id="a" factory-method="getAInstance"/>

**public** **static** A getAInstance() {

**if**(*obj*==**null**) {

*obj*= **new** A();

}

**return** *obj*;

}

//<bean factory-bean="a" factory-method="getBInstance" id="b"/>

**public** B getBInstance() {

**return** **new** B();

}

}

**class** B{

}

//A obj= new A(); B bObj=obj.getInstance();

BeanPostProcesor