***Date : 12-08-2018***

**Spring Core Module**

|  |  |  |
| --- | --- | --- |
| 1. | **What is Dependency Injection?** |  |
| 2 | **What is the difference between FileSystemResource and ClassPathResource?** |  |
| 3 | **Workflow** |  |
| 4 | **Collaboration**  **How many ways we can use dependent class logic** |  |
| 5 | **Design pattern** |  |
| 6 | **Resolving constructor parameters** |  |
| 7 | **Cyclic dependency** |  |
| 8 | **Application of setter and constructor injection** |  |
| 9 | **Inner beans** |  |
| 10 | **Bean Inheritance** |  |
| 11 | **Collection Merging** |  |
| 12 | **Bean alias** |  |
| 13 | **Dependency Lookup** |  |
| 14 | **Singleton java class** |  |
| 15 | **Bean Scope** |  |
| 16 | **Bean Wiring** |  |
| 17 | **Dependency Check** |  |
| 18 | **Nested Bean Factories** |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

## Spring Definition

What is **dependency lookup?**

What is **dependency injection?**

What is **POJO, POJI, Java Bean, Spring bean?**

## What is Dependency Injection?

What are the types?

1. Setter injection
2. constructor injection
3. interface injection
4. method injection
5. Lookup method injection
6. ***Setter Injection***
7. Spring container calls 0-param constructor (to create an obj of target object) and calls setter methods to perform setter injection.
8. Here target class object is created first, then dependent class object is created, then setter injections are done
9. ***Constructor Injection***
10. Spring container call parameterized constructor to create object of the spring bean and inject object to target object.
11. Here dependent class object is created first then target class object is created.
12. ***If both injections are done (setter as well as constructor injection), then setter injection will dominate constructor injection because setter methods execute after construction injection and overrides the values injected by constructor injection.***
13. ***Setter injection – 0- param constructor – to create bean class object***

***Constructor injection – param constructor – to create bean class object***

***Date : 20-08-2018***

**How many ways to get target class object from container?**

WishGenerator wish=factory.getBean("wish", WishGenerator.class); //no type casting

WishGenerator wish = (WishGenerator) factory.getBean("wish");//type casting required

## Workflow

* - - -

## What is the difference between ClassPathResource and FileSystemResource?

**ClassPathResource** can locate the spring configuration file from directory or jarfiles that are added to classpath/buildpath **and**

**FileSystemResource** locate the spring configuration file from given path. We may give absolute / relative path.

## Collaboration, tight and loose coupling

What is collaboration?

Keeping spring beans in dependency is called as collaboration

## How many ways target class can use logics of dependent class? What are their limitations

|  |  |  |
| --- | --- | --- |
| 1. Use dependent class object in target class *(composition)* |  |  |
| 1. Create dep class obj in Factory class and make target class to access dep class obj through that factory class *(composition)* |  |  |
| 1. Get dep class obj from Jndi registry *(dependency lookup)* |  |  |
| 1. Extend target class from dep class *(inheritance)* |  |  |
| 1. Make underlying container inject dep class obj in target class *(dependency injection)* |  |  |

What is the solution for getting loosely coupled?

It is recommended that all dependent class should implement one common interface having common methods so that we will use that common interface in our target class.

## Design Pattern

What is design pattern?

It is set of rules to solve reoccurring problems in application development.

BeanFactory works on **FactoryDesign** pattern.

It is recommended to use **strategy design pattern** whenever **we develop the cases having dependencies**.

***strategy design pattern***

It consists of 3 rules.

1. **prefer composition over inheritance**
   1. multiple inheritance is not possible in java
   2. if two classes are in IS-A relationship and they have overridden methods, if super class method’s return type/argument is changed. It will affect al its child classes. That means code becomes easily breakable.
2. **always code to interface but not to implementation class**

Example 1

Class A {}

Class B {}

Class Target{

//dep class

A a=new A();

OR

B b=new B();

}

Tight coupling between target class and dependent class

interface X{}

Class A implements X{}

Class B implements X{}

Class Target{

//dep class

**X x=new A(); OR X x=new B();**

**}**

Loose coupling between target class and dependent class

We can achieve loose coupling if we go for interface

1. **code must be available for extension but not for modification**
   1. by keeping methods of dependent class final we cannot override them

***Date : 20-08-2018***

## Resolving Constructor params through name, type, index

Resolving constructor injection params by

1. **Type**: We can specify type of the parameter*. if multiple params are there of same type, ambiguity problem may come. So use name or index as a solution*
2. **Index**: here we need to remember the index of the param, otherwise problem will come (index starts with 0 always)
3. **Name**: it is the best approach to follow.

***There is no need of resolving constructor parameters, if they are configured in the order they are originally. If you want you can resolve by above type.***

*In Spring 3.x, IOC container can’t get param name of constructor directly until the param name are configured by using* ***@ConstructorProperties()***

@ConstructorProperties({ "id", "name", "city" })

**public** TargetClass(String id, String name, String city) {

**super**();

**this**.id = id;

**this**.name = name;

**this**.city = city;

}

***In spring 4.x, they are recognized directly.***

## Cyclic dependency

If two beans are dependent on each other then we can say that they are in cyclic dependency. This cyclic dependency is only possible in setter injection not in constructor injection.

***Date : 12-08-2018***

## When should we should we use setter and constructor injection?

If all properties of bean class want to participate in dependency injection, then we should go for construction and out of all properties, we want some properties to participate in dependency injection, then we should go for setter injection.

## Difference between setter injection and constructor injection

|  |  |
| --- | --- |
| **Setter injection** | **Constructor Injection** |
| Used when few properties are to be injected | Used when all properties to be injected |
| Target bean class object is created first, then dep class obj | Dep class object class is created first, then target class obj |
| Hence slower injection | Faster injection |
| Container calls 0-param constructor | Container calls param constructor |
| Cyclic dependency is possible | Not possible |
| <property> tag is used | <constructor> tag is used |
|  |  |

## Inner bean

### What is inner bean?

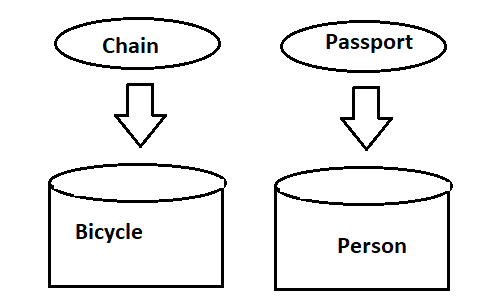
When a bean is configured as inner bean of other bean then it is called as inner bean. Its no way related to inner classes. It is performed by **<bean>**tag inside **<property> tag or <constructor> tag.**

### How to declare it as a inner bean?

By declaring <bean> tag inside <property> tag. Id attribute inside <bean> tag is optional.

### When to use inner bean?

***When we have to make a bean as dependent bean to another bean only once or when a bean does not need to be shared to other classes, we should go for inner beans.*** For ex:



### Can we declare bean without id?

*Yes, because spring container generated default bean ids like* ***<pkg><classname>#number***

### Why it is possible to configure inner bean without bean id?

*By its definition inner ban will not be shared outside its enclosing bean. Inner bean id will be ignored by Spring. So there is no requirement to provide bean id to it.*

### Limitation of inner beans

1. ***Inner beans cannot be used as dependent bean for multiple other beans.***
2. ***We cannot access inner bean object from client application.***

## Null Injection

### What is Null injection?

Injecting null value to the Object/String type property of constructor injection is called as Null Injection.

### How to do it?

By using <null/> inside <constructor> tag/<property> tag

### Why to do it?

In setter injection, it’s not necessary to inject all the properties and we can pass null value to String or object type property. But in constructor injection, all properties must be configured and we cannot pass null value directly other we will get **UnsatsifiedDependancyInjectionException.** So to overcome this, we should inject null value in constructor-arg by using null injection.

**public** **class** TargetClass {

**private** DependentClass depClass;

**private** String id;

**private** String name;

**private** String city;

//par-constructor

//setter getters

}

Public class DependentClass{

**private** String hobbyname;

}

**CONSTRUCTOR INJECTION**

<bean id=*"client1"* class=*"com.bs.TargetClass"*>

**<constructor-arg><null/></constructor-arg>**

<constructor-arg value=*"123"* />

<constructor-arg value=*"Ankur"* />

<constructor-arg value=*"Amrawati"* />

</bean>

**SETTER INJECTION**

<bean id=*"client1"* class=*"com.bs.TargetClass"*>

<property name=*"depClass"* ref=*"service1"* />

<property name=*"id"* value=*"23"* />

**<property name=*"name"*>**

**<null /></property>**

<property name=*"city"* value=*"Morshi"* />

</bean>

## Bean Inheritance

### What is bean inheritance?

Reusing the existing configuration property of parent bean in child bean is called as **bean inheritance**.

### When?

1. When we want to configure same class multiple times as a spring bean, so instead of declaring all properties again, it is easy to reuse the existing bean properties configuration.
2. When we want to use configuration properties of one bean inside other bean.

<bean id=*"pul1"* class=*"com.bs.Pulsar"*>

<property name=*"id"* value=*"12"*/>

<property name=*"engineCode"* value=*" EN13"*/>

<property name=*"color"* value=*"white"*/>

</bean>

<bean id=*"pul2"* class=*"com.bs.Pulsar"*>

<property name=*"id"* value=*"12"*/>

<property name=*"engineCode"* value=*"EN13"*/>

<property name=*"color"* value=*" white "*/>

</bean>

Here we are using color, engineCode again.

Better to go for Bean Ineritance.

**Solution 1 : take some properties from pul1**

<bean id=*"pul1"* class=*"com.bs.Pulsar"*>

<property name=*"id"* value=*"13"*/>

<property name=*"engineCode"* value=*"EN12"*/>

<property name=*"color"* value=*"white"*/>

</bean>

<bean id=*"pul2"* class=*"com.bs.Pulsar"* **parent=*"pul1"*** >

<property name=*"id"* value=*"12"*/>

</bean>

**Solution 2 : take common properties from baseConfiguration**

<bean id=*"parentConf"* class=*"com.bs.Pulsar"* abstract=*"true"*>

<property name=*"engineCode"* value=*"EN12"*/>

<property name=*"color"* value=*"white"*/>

</bean>

<bean id=*"pul2"* class=*"com.bs.Pulsar"* **parent=*"pul1"*** >

<property name=*"id"* value=*"12"*/>

</bean>

<bean id=*"pul2"* class=*"com.bs.Pulsar"* **parent=*"pul1"*** >

<property name=*"id"* value=*"13"*/>

</bean>

### How?

1. Declare one bean which has common property values by using <bean> tag and make it as abstract by using **<abstract=”true”>** attribute.
2. Now declare child beans by <bean> tab which will inherit the common property value from an abstract bean. For this we have to add **parent=”abstractbeanid”** attribute. (we can make as many child beans as we want.)

### Imp points

1. **Here parent bean class type may be different from child bean class type but all the properties in parent bean must be present in child bean.**
2. **When we inherit one bean from another bean, only parent bean property value will be copied to child bean property value but physically classes never get extended.**
3. **It is recommended to take one of the beans as abstract bean which will contain common values that should be inherited to child bean. As it is abstract we never modify property values unless it has to be affected to all.**
4. **When we define a bean as abstract bean, the class will not become abstract, only current bean definition will become abstract means spring container never going to create an object of that bean.**
5. **Child bean property can overwrite parent bean properties.**

In bean inheritance, we can reuse the parent bean configuration properties and overwrite also. ***But we cannot merge/add any property to the parent bean configuration bean property***. For that we should go for ***Collection merging.***

## Collection Injecton

|  |  |  |
| --- | --- | --- |
| **Simple property** | **Tag for dependency injection** |  |
| Simple property | Value attribute or <value> tag |  |
| Reference property | Ref attribute or <ref> tag |  |
| Array | <list> | Allow duplicates |
| List | <list> |
| Set | <set> | Doesn’t allow duplicates |
| Map | <map> |  |
| Properties | <props> |  |

***Date :31-08-2018***

Various ways to configure map type property

<beanid=*"faculty"*class=*"com.nt.beans.Faculty"*>

<propertyname=*"facultySubjects"*>

**Type1**

<map>

<entry>

<key>

<value>Natraj</value>

</key>

<value>AJ</value>

</entry>

**Type2:**

<entrykey=*"HK"*value=*"CJ"*/>

**Type3:**

<entrykey=*"murali"*>

<value>oracle</value>

</entry>

</map>

</property>

**Type4: when to configure ref type property**

<propertyname=*"batchDates"*>

<map>

<entry>

<key><value>AJ</value></key>

**<refbean=*"dt1"*/>**

</entry>

<entrykey=*"CJ"*value-ref=*"dt2"*></entry>

</map>

</property>

</bean>

### Collection Merging using merge="true" using XML

### What s Collection merging?

### How to do it?

### *When to use Collection merging? (2 uses)*

***If target class is having collection type property***

***If we have to inherit parent type property (using merge)***

* Merging the parent bean property collection with child bean property ***collection.***

In spring, java collections can be injected in bean using application xml. ***According to the need, we can merge collection data while creating bean***. There will be a parent and child bean. Child will inherit the parent data. It is done by the attribute merge="true". This attribute is available in all collection tags like <list/>, <set/>, <map/> and <props/> . The child bean will use parent attribute to inherit parent bean data. The child bean collection tag will use merge="true" attribute to inherit parent collection elements. When we access child bean, we will get elements of child as well as parent bean collection. This is collection merging of beans in spring.

Rules

* **Collection merging only comes in picture when there is bean inheritance.**
* Parent property collection type must be same as child property collection type.
* Parent property collection generic type must be same as child property collection generictype.

**[If we have to inherit some other class property which is of same type, so it is better to go for collection merging having bean inheritanceinstead of again creating bean id and all that**]

## Bean Alias

### What is bean alias?

Bean alias is nothing but the alternate name/id to bean id. It provides alternate name to bean id.

### When to use?

1. Sometimes bean class name is taken as bean id which may be too lengthy, so to make it as a shorter version, bean aliasing is used.
2. To provide loosely coupling in bean names

### How to declare?

Prior to Spring 2.x, **name attribute** is there to provide alternate name but from Spring 2.x onwards, **<alias> tag** is used to do this.

## Dependency lookup/Dependency pull

### What is dependency lookup?

When target class/main class writs code to search and get the dependent objects is called as dependency lookup/pull.

### When /why to use it?

If we want ***dependent object in specific methods of target class***, then we should go for ***dependency lookup/pull***. If we want dependent object in all the methods of target class then go for ***dependency injection.***

### How to use it?

By creating IOC container in specific method of target class to get dependent objects we can perform dependency lookup.

### Dependency Lookup problem:

In this we create IOC container in target class method once again in addition with IOC container created in main class. So creating two IOC container is meaningless. So to avoid this, we can send IOC container created in main class as an argument to the method of target class so that we can simply reuse the IOC container in target class method. But this is th temporary solution because creating a business method with IOC container as an argument makes an application as invasive application which is against non-invasive behaivior.

## Id-ref

In dependency injection/ dependency lookup, if we are sending the bean id to the value attribute of <property> tag, **it will not check whether the bean class is configured with given bean id or not**. It simply consider it as a String value. So latter if we change the value of the value attribute. It won’t check whether bean is configured or not. It will inject it as a String value only which may lead to an ***NoSuchBeanDefinitionException*.** So to avoid this and to inject a bean id only we must use **<idref>**tag having bean attribute.

<ref> tag 🡪referring an object

<idref> tag 🡪ref to an id of another object

## Factory methods

### What is factory methods?

The method which is used to create and return same class object or different class object is called as factory method.

### What are its types?

Static factory method, instance factory method(both methods can return same or different class object)

### When/why to use?

*Static factory method:-*

1. Sometimes class cannot be instantiate by using new operator. In that class, some static methods are there to create that class object only, that time we should go for **static factory method injection***.*

*Instance factory method:-*

1. Sometimes the creation of some class object involves few steps or few operation to be performed. So these steps are coded in a method of some class which is called as ***Service Locator class***. These are the classes know how to create the object of the target bean.
2. So to create our clas
3. s object you need to instantiate the service locator class on which in turn you will call the method who will create our class object. **Instance factory method injection** allows to instantiate the target class object by calling method on our class (Service locator) class.

### How to use?

***Static factory method:-***

1. Declare that class using **<bean> tag**, use **factory-method** attribute to specify the static factory method which is going to create that class object.

***Instance factory method:-***

1. Declare our service locator class as a spring bean on which we will call factory method
2. Configure the target bean which has to be created by service locator class and declare **<bean> tag**. Use **factory-bean attribute** which is our service locator bean**, factory-method attribute** which is our factory method in service locator class which will create our class (target class) object.

* Use ***constructor-arg*** elements to specify **arguments to the factory method**, if it takes arguments.

### Limitation of factory methods

Factory methods should take zero arguments, then only we can configure as a factory method in configuration file. If our factory method is taking parameters then our class cannot be instantiated by user supplied inputs, then object will be created with default values.

## Singletone class

### What is singletone class?

The class which allows to create single object only er jvm is called as Singletone java class.

### Why to use it?

1. When class is not having any state
2. When class si having read only sharable state
3. When class is having huge amount of states and having multiple write operations in synchronized manner.

* **Instead of creating multiple object of class, create one object and access it multiple times, that time create a class as singletone class.**
* **If we just create a single object of a class and that class allows us to create multiple objects does not mean it is a singletone class.** Ex: our servlet class allows us to create multiple class but most of the time servlet container single servlet class

### How to use it?

## Bean Scope

### What is bean scope?

It is a logical boundary within which bean is deployed and know to an application.

How many bean scopes are there**?**

1. **Singleton scope (default scope)** – IOC container returns single bean class object with same bean id for every factory.getBean() method calls
2. **prototype**– IOC container returns creates bean class object for every factory.getBean() method call
3. **request**– creates bean class object specific to the http request
4. **session**– creates bean class object visible throughout the session

### How spring container will return same object when the scope is singletone?

*IOC container internally maintains HashMap whose key is spring bean id and values is the spring bean class object as a value. For next factory.getBean() call, it will check that object in HashMap first. If it is available, it will return this available object otherwise it will create a new object. But in prototype scope, IOC container will not maintain HashMap so everytime new object will be returned.*

***Singletone Scope*** *– IOC container keeps created bean class object in HashMap where key=bean id and value=bean class object and uses this object across multiple method calls*

***Prototype Scope*** *– IOC container doesn’t keep the object in HashMap, it creates it and passes every time a new instance*

***Request scope*** *– IOC container keep object in request attribute.*

***Session scope*** *– IOC container keeps object in session object.*

### Why they exist?

### How to configure?

Using scope attribute under <bean> tag.

**NOTE : singletone scope never makes a bean class as singletone java class, it simply bring the effect of singletone java class.**

## Bean Wiring

### What is bean wiring?

Configuring spring bean using

primary=*"true":*Specifiesthat this bean should be given preference when multiple candidates are qualified to autowire a single- valued dependency. If exactly one 'primary' bean exists among the candidates, it will be the autowired value

### Types of bean wiring

**autowire=”byName”**

**Here spring container searches the property name which has setter methods on the target class, finds bean configuration whose name/id is same as name of the property and performs the injection by calling setter methods.**

**OR**

***Here container searches for the property whose name is exactly equal to the bean in/name configured as a dependent bean in spring bean configuration file. When target bean has a property whose name is exactly equal to the dependent bean id/name, it performs injection using setter methods.***

**Conditions:**

**If multiple beans are**

**autowire=”byType”**

**autowire=”constructor”**

**autowire=”no”**

**autowire=”autodetect”**

### How to do it?

Keep**autowire** attribute in target class.

**Primary=”true” autowire-candidate=”false”**

## Dependency Check

### What is dependency check?

Providing constraints to setter injection to inject all the dependent object into to target class is called as Dependency check.

### Why/when dependency check?

So that all dependent object should be injected.

### How to do that?

By using dependency-check attribute in the target bean configuration.

Syntax:

dependency-check=”value”

values are:

1. ***all*** :checks whether all **simple**&**reference** properties are configured for setter injection or not.
2. ***none*** : dependency check is disabled.
3. ***simple*** : checks whether all **simple** properties are configured for setter injection or not.
4. ***object*** : checks whether all **reference** properties are configured for setter injection or not.

## Nested Bean Factories

**What isNested bean factories?**

Using parent containers beans inside child container beans is called as nested bean factories.**OR**Using beans of one container inside bean of other container called as nested bean factories.

**Why Nested bean factories?**

In real time projects, multiple layers will be there having multiple beans. So instead of having only one container for all layers, it is recommended to have **cotainer per layer.** It will be possible by taking nested containers like parent container and child containers.

**How Nested bean factories are used?**

Tier components

1. business tier components – components that deals with business logic and persistent logic
2. presentation tier component – components that takes input, displaying output by handling req, response

As presentation tier uses business logic, so manage business tier components in parent IOC container and presentation tier component in child IOC container.

**Event handling**

* **Event** means action raise on a component /object, **event handling**mean executing a logic when event is raised.
* For this four actors are required
  + Event class
  + Event Listener
  + Event Handling method
  + Source object
* We perform event handing on ‘Application Context container” to keep track of when the container is started and stopped.
* It allows programmer to evaluate spring performance.

Steps to do:

1. Take any application having application context support, add a class implementing **ApplicationEvent** and implement **onApplicationEvent**(ApplicationEvent ae) method
2. Configure event class in applicationContext.xml file

For example:

**publicclass** EventClass **implements** ApplicationListener<ApplicationEvent> {

@Override

**publicvoid**onApplicationEvent(ApplicationEvent ae) {

**long**start = 0;

**long**end = 0;

**if** (ae.toString().indexOf("Refreshed") != -1)

start = System.*currentTimeMillis*();

**else**

end = System.*currentTimeMillis*();

System.***out***.println("Container acting time " + (end - start));

}}

**Exceptions occurred in Spring**

|  |  |
| --- | --- |
| BeanCreationException | When dependent class object is not injected in target class. When we don’t provide dep class object through <constructor-arg> tag. |
| UnsatsifiedDependancyInjection | When during configuring target class by constructor injection,any one property of the target class is not configured |
| NoSuchBeanDefinitionException |  |
| UnsatisfiedDependencyException | Exception thrown when a bean depends on other beans or simple properties that were not specified in the bean factory definition, although dependency checking was enabled |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |