SPRING core

Spring CORE modules

* Spring core
* Spring JDBC
* Spring ORM

# Spring CORE Provides basic fundamental support for SPRING.

****

# CORE:

## BEANS:

Fundamental part of Spring core

Provides IOC and dependency feature

## CONTEXT:

It inherits multiple features from beans ie: factory design feature

internationalization

event propagation

transparent creation of context

resource loading

## spEL: (Spring Expression Language)

Power to Queries and manipulates values of object at runtime.

## AOP:(Aspect oriented programming)

Allows us to define method interceptors(Like some stuff we wanted to before/after method)

We can define point cuts

We can cleanly decouple the code.

Aspect:

Instrumentation:

Provides class instrumentation support

Messaging:

Serves a foundation for messaging based application.

We can map the message to our methods using appropriate annotation

# \*DATA\*

## JDBC:

* Provides an extra JDBC layer to ease out database connectivity
* Spring ORM (Object relational mapping) :
* Integration for Hibernate etc

## Spring OXM:

* Provides an object XML mapping
* Castor,X Stream, JAXB

## JMS

* + Java messaging service
  + Produces and consumes messages

# WEB

web oriented integration feature

helpful for rest or web-based app

contains http client and web related stuffs

## WEB:

## SERVLET:

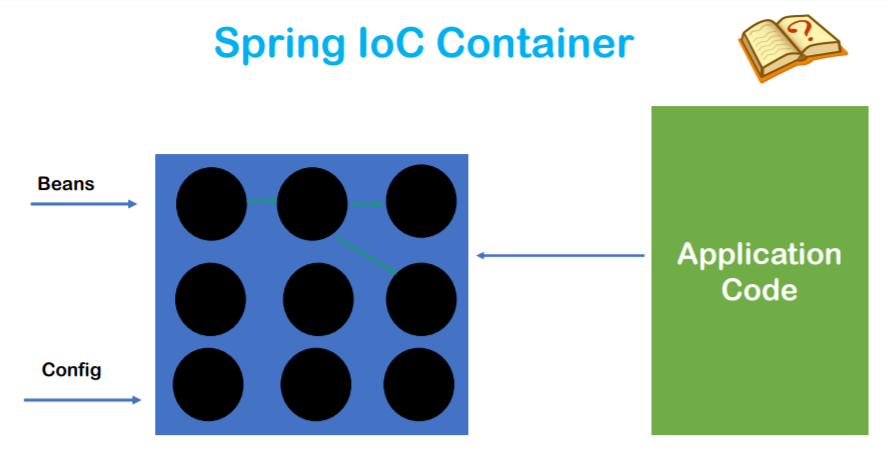
## PORTLET: useful for portlet-based app

## WEB SOCKET:

# TEST

Provides J Unit and test NG for unit / integration testing

provides mock objects for testing in isolation.



SPRING IOC Container:

Its a predefined component available with SPRING

Holds the created object in the memory and helps in DI

Holds account for life cycle of Object

Provides ready to use object

We have to instruct what beans or POJO classes it has to manage and the configuration XML file

we will instruct it which

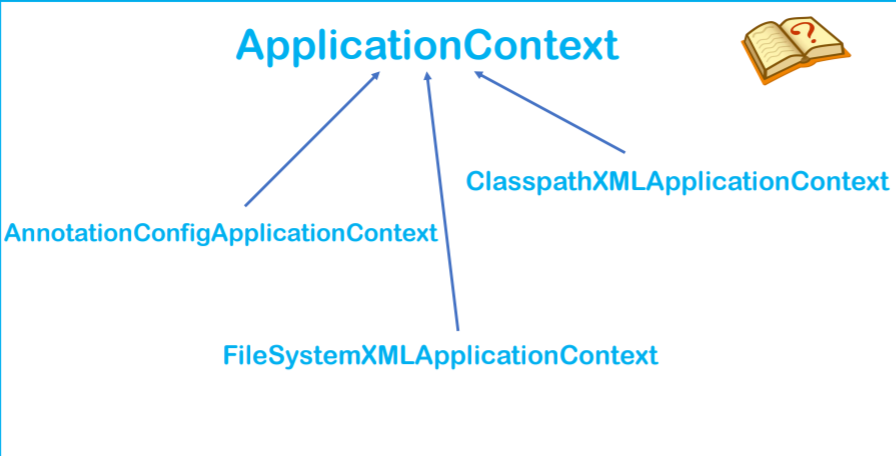
object has a dependency which objects

and it will inject at suitable places accordingly.

Need 2 things

POJO

XML speaking about dependency



APPLICATION CONTEXT:

Its an interface and represents IOC

It extends bean factory

It represents our IOC container

ClasspathXMLApplicatonContext:

It searches XML config in JAVA class path

AnnotationConfigAppicationContext

Searches for annotation used on the bean

FileSystemXMLApplicatonContext

Searches file system for config file

##Lecture 5



IOC will create the object of Address and inject all its state

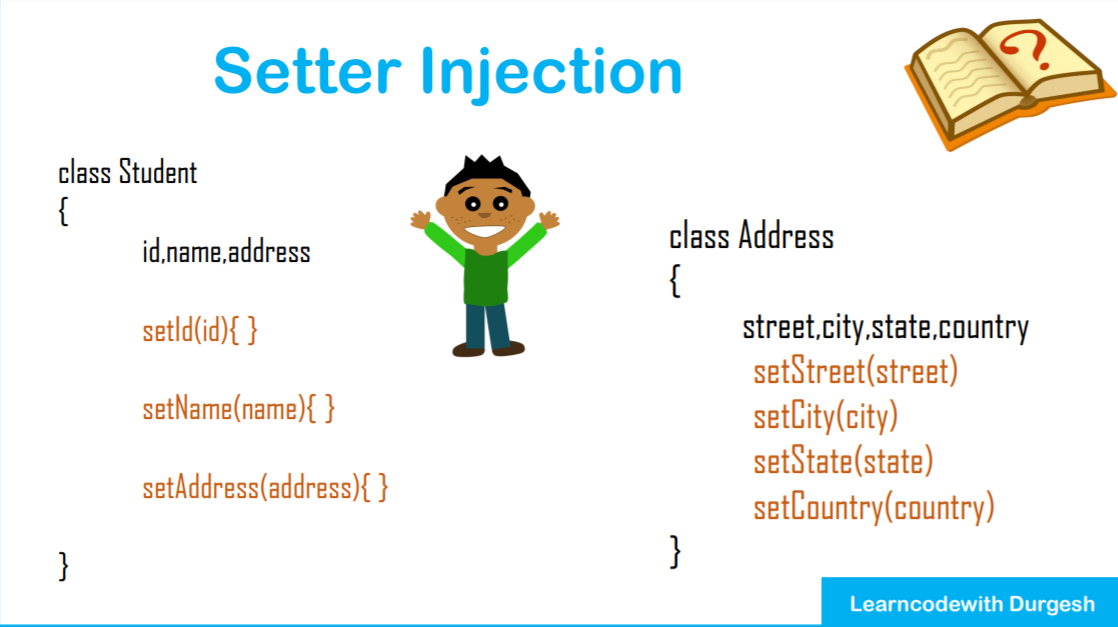
It will also create object of Student and inject its state and the most important it will inject the Previous object of address at RUNTIME.

After this we can ask IOC for the beans of Student with all its dependencies full filled.

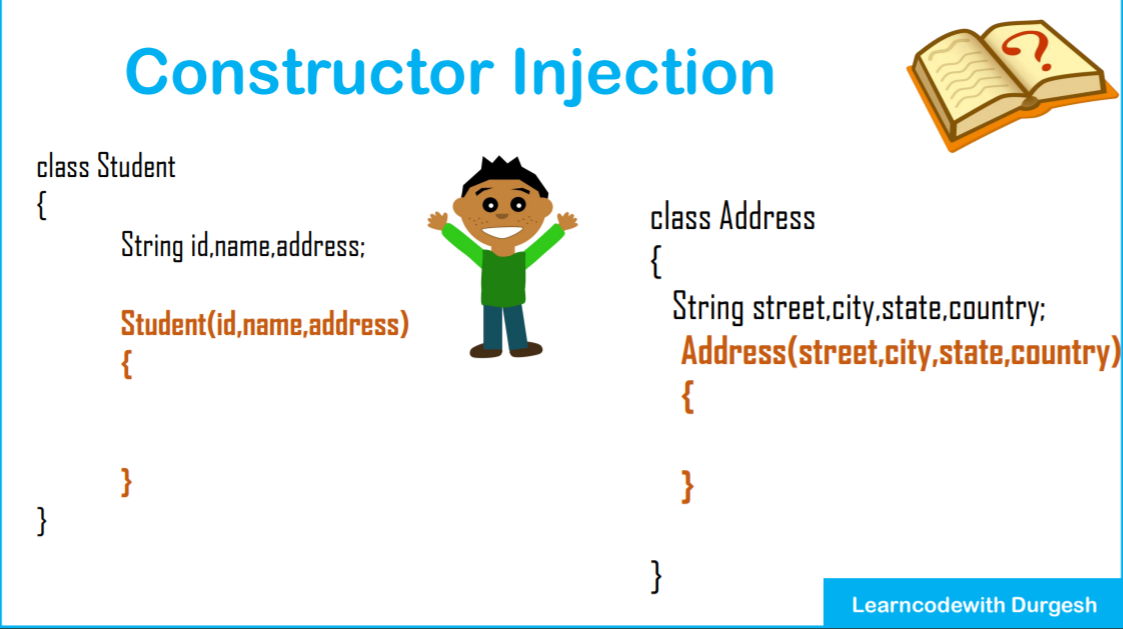
Dependency can be injected in 2 ways:

Setter Injection

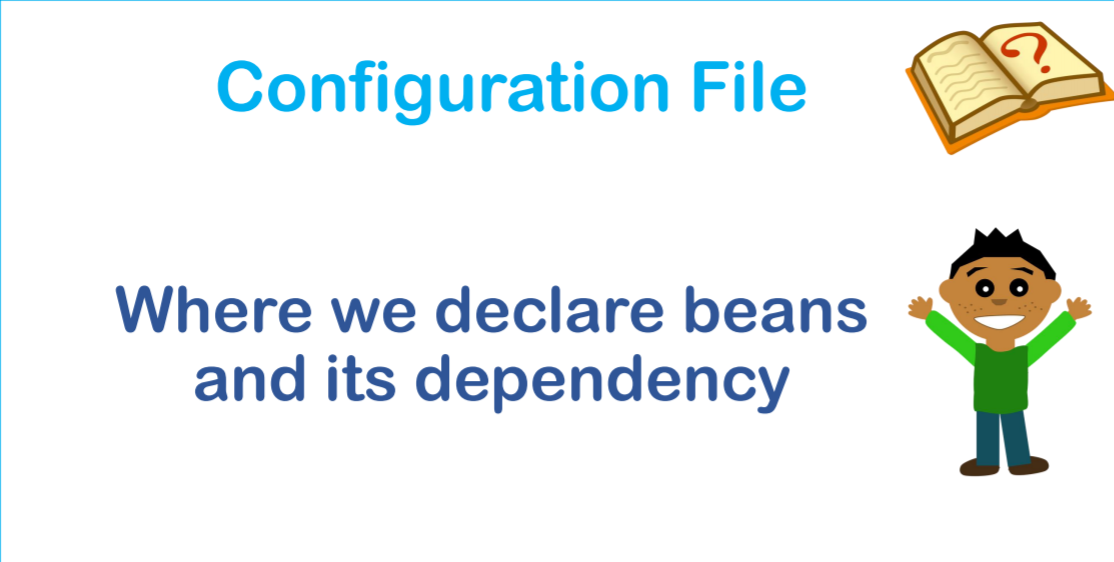
Constructor Injection



While creating the object it will call the setters while creating object. Meanwhile updating the state(instance variables).



In this case while creating object , it will use constructor meanwhile updating the values

\

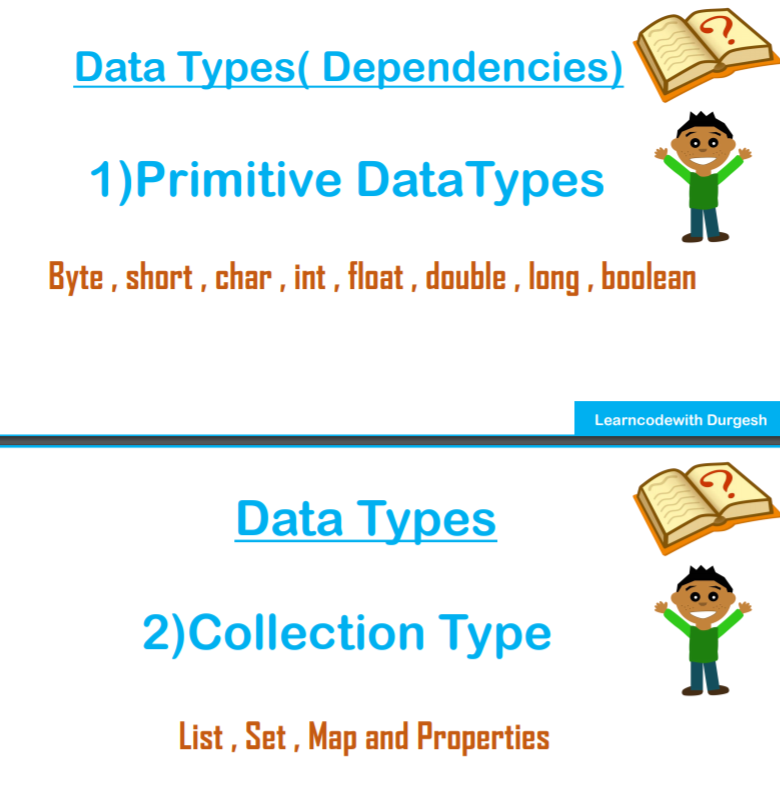
We will provide instructions for Spring (IOC) of the classes(beans and its dependencies)

1st tag

<beans>

<bean> </bean>

</beans>



While injecting it sees the datatypes

* byte
* short
* char
* int
* long
* float
* double
* boolean

It also supports collection types

* List
* Set
* Map
* Properties

Can also inject Reference type

* Address

##Lecture 6,7

Software Config

Eclipse/Intellij

Tomcat

Mysql

sqlyog / workbench / phpadmin for mysql

STEPS

Create a Maven Project

New -> Maven Project -> maven-archtype-quickstart

Group ID: com.springcore

Artifact ID: springcore

version: 0.01-spanshot

package com.springcore

Fullfilling the Maven dependency for spring

Adding from MAVEN REPOSITORY

Spring Core (5.3.3)

Spring Context (5.3.3)

paste the dependency tag in pom.xml

create beans (java pojo)

**STUDENT CLASS**

* attributes
* getter setter
* constructors
* toString override

create config.xml and declare all the instructions for string

*We are also using Property injection using* ***P*** *SCHEMA*

<?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:context="http://www.springframework.org/schema/context"

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

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

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

*<!-- this is our 1st bean -->*

<bean class="com.spring.core.springcore.Student" name="student1">

<property name="studentId" value="6057"/>

<property name="studentName">

<value>Suman</value>

</property>

<property name="studentAddress">

<value>Bangalore</value>

</property>

</bean>

*<!-- this is our 2nd bean using inline property tag -->*

<bean class="com.spring.core.springcore.Student" name="student2">

<property name="studentId" value="6058"/>

<property name="studentName">

<value>Suphlay</value>

</property>

<property name="studentAddress">

<value>Kolkata</value>

</property>

</bean>

*<!-- this is our 3rd bean using P Schema(properties) and it has to be included in the schema tag-->*

<bean class="com.spring.core.springcore.Student" name="student3" p:studentId="6059" p:studentName="Ujjwal" p:studentAddress="Hooghly"/>

</beans>

setter injecton ready

app.java

package com.spring.core.springcore;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

/\*\*

\* Hello world!

\*

\*/

public class App

{

public static void main( String[] args )

{

System.out.println( "Hello World!" );

// we will not use new keyword to drectly make objects

// if one .xml pass it asa string for multipasss it as a string aray

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

// if stored in package use full path for .xml file

Student s1 = (Student) context.getBean("student1");

System.out.println(s1);

Student s2 = (Student)context.getBean("student2");

System.out.println(s2);

Student s3 = (Student)context.getBean("student");

System.out.println(s3);

}

}

##Lec 8

We will inject collection type (List,Set,Map and Properties)

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

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

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

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

*<!-- this is our 1st bean and we are injecting collections -->*

<bean name="emp1" class="com.spring.core.collections.Employees">

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

<property name="phones">

<list>

<value>9563265148</value>

<value>9563265148</value>

<value>9563265148</value>

<null/>

</list>

</property>

<property name="pin">

<value>828130</value>

</property>

<property name="tags">

<list>

</list>

</property>

<property name="address">

<set>

<value>Delhi</value>

<value>Lucknow</value>

<value>Kanpur</value>

<value>Delhi</value>

</set>

</property>

<property name="course">

<map>

<entry key="Java" value="3"/>

<entry key="Python" value="4"/>

<entry key="JS" value="6.5"/>

</map>

</property>

<property name="general">

<props>

<prop key="gender">male</prop>

<prop key="age">24</prop>

<prop key="passportAvailbility">YES</prop>

</props>

</property>

</bean>

</beans>

CollectionTester.java

**package** com.spring.core.collections;

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

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

**public** **class** CollectionTest {

/\*\*

\* Hello world!

\*

\*/

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

{

System.*out*.println( "START\n" );

// we will not use new keyword to drectly make objects

// if one .xml pass it asa string for multipasss it as a string aray

ApplicationContext context = **new** ClassPathXmlApplicationContext("com/spring/core/collections/collectionconfig.xml");

// if stored in package use full path for .xml file

Employees emp1 = (Employees)context.getBean("emp1");

System.*out*.println(emp1.getName());

System.*out*.println(emp1.getPin());

System.*out*.println(emp1.getTags());

System.*out*.println(emp1.getPhones());

System.*out*.println(emp1.getAddress());

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

System.*out*.println(emp1.getCourse());

System.*out*.println(emp1.getGeneral());

System.*out*.println( "END\n" );

}

}

##Lecture 9

We will inject Reference Type

Object A has a dependence of B

so A is a dependent object depending on B

*XML coment in eclipse* ***ctrl+shift+c***

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

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

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

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

*<!-- this is our B bean -->*

<bean class="com.spring.core.reference.B" name="bref">

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

</bean>

*<!-- this is for our 1st A bean -->*

<bean class="com.spring.core.reference.A" name="aref1">

<property name="x" value="24"/>

*<!-- <property name="obj"> -->*

*<!-- <ref bean="bref"/> -->*

*<!-- </property> -->*

*<!-- new way -->*

<property name="obj" ref="bref"/>

</bean>

*<!-- using p Schema ,we are creating 2nd A bean -->*

<bean class="com.spring.core.reference.A" name="aref2" p:x="25" p:obj-ref="bref"/>

</beans>

TestReference.java

**package** com.spring.core.reference;

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

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

**public** **class** TestReference {

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

// **TODO** Auto-generated method stub

ApplicationContext context = **new** ClassPathXmlApplicationContext("com/spring/core/reference/refconfig.xml");

A temp = (A)context.getBean("aref1");

System.*out*.println(temp.getX());

System.*out*.println(temp);

System.*out*.println(temp.getObj());

System.*out*.println(temp.getObj().getY());

A temp2 = (A)context.getBean("aref2");

System.*out*.println(temp2);

}

}

##Lecture 11, 12

Constructor Injection

constructor can utilize p-schema rather than c-schema for property injection

Removing ambiguity and specifying the order of parameters

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

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

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">

*<!-- this is our 1st bean for Certi using C-schema-->*

<bean class="com.spring.core.ci.Certi" name="cert" c:name="JAVA-OCJB">

*<!-- <constructor-arg value="Spring WEB"/> -->*

</bean>

*<!-- this is our 1st bean for Person -->*

<bean class="com.spring.core.ci.Person" name="person1">

*<!-- <bean class="com.springcore.ci.Person" name="Person" -->*

*<!-- c:name="Suman" c:age="20" c:certi-ref="certi"> -->*

<constructor-arg>

<value>Suman</value>

</constructor-arg>

*<!-- <constructor-arg value="18"/> -->*

*<!-- It will automatically convert compatible String to int. -->*

*<!-- but we can explicitly specify also the type -->*

*<!-- its a good habit to define the type explicitly -->*

*<!-- because it will avoid ambiguity in multiple constructors -->*

<constructor-arg value="18" type="int"/>

*<!-- we also cannot use p-type instead we use c-type schema -->*

<constructor-arg ref="cert"/>

<constructor-arg type="java.util.List">

<list>

<value>Python</value>

<value>JAVA</value>

<value>JS</value>

</list>

</constructor-arg>

</bean>

<bean class ="com.spring.core.ci.Addition" name="add">

<constructor-arg value="40" type="int" index="1"/>

<constructor-arg value="60" type="int" index="0"/>

*<!-- order can be interchanged via index -->*

</bean>

</beans>

Person.java

**package** com.spring.core.ci;

**import** java.util.List;

**public** **class** Person {

**private** String name;

**private** **int** age;

**private** Certi certi;

**private** List<String> listOfCourses;

**public** Person(String name, **int** age,Certi certi,List<String> listOfCourses) {

**super**();

**this**.name = name;

**this**.age = age;

**this**.certi = certi;

**this**.listOfCourses = listOfCourses;

System.*out*.println("Person constructor called");

}

@Override

**public** String toString() {

**return** "Person [name=" + name + ", age=" + age + ", certi=" + certi + ", listOfCourses=" + listOfCourses + "]";

}

}

Certi.java

**package** com.spring.core.ci;

**public** **class** Certi {

**private** String name;

**public** Certi(String name) {

**super**();

**this**.name = name;

}

@Override

**public** String toString() {

**return** "Certi [name=" + name + "]";

}

}

Addition.java

**package** com.spring.core.ci;

**public** **class** Addition {

**private** **int** a;

**private** **int** b;

/\*

\*Here ambiguity arises if we have.t define type in .xml

\*due to ambiguity the top order constructor

matching the parameter is getting executed

\*First priority goes to String irrespective of order

\*Then the order if it doesn't matches

\*

\*/

**public** Addition(**double** a, **double** b) {

**super**();

**this**.a = (**int**)a;

**this**.b = (**int**)b;

System.*out*.println("constructor : double | double");

}

**public** Addition(**int** a, **int** b) {

**super**();

/\*

\* Spring is so intelligent that it can match the best constructor from the

\* first explicit type="int only"

\*/

**this**.a = a;

**this**.b = b;

System.*out*.println("constructor : int | int");

}

**public** Addition(String a, String b) {

// highest priority to String

**super**();

**this**.a = Integer.*parseInt*(a);

**this**.b = Integer.*parseInt*(b);

System.*out*.println("constructor : String | String");

}

**public** **void** doSum() {

System.*out*.println("a = "+a+" | b = "+b);

System.*out*.println("SUM ="+(**this**.a+**this**.b));

}

}

TestCI,java

package com.spring.core.ci;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class TestCI {

public static void main(String[] args) {

// TODO Auto-generated method stub

ApplicationContext context = new ClassPathXmlApplicationContext("com/spring/core/ci/ciconfig.xml");

Person p1 = (Person)context.getBean("person1");

System.out.println(p1);

Addition a1 = (Addition)context.getBean("add");

a1.doSum();

}

}

o/p

Person constructor called

constructor : int | int

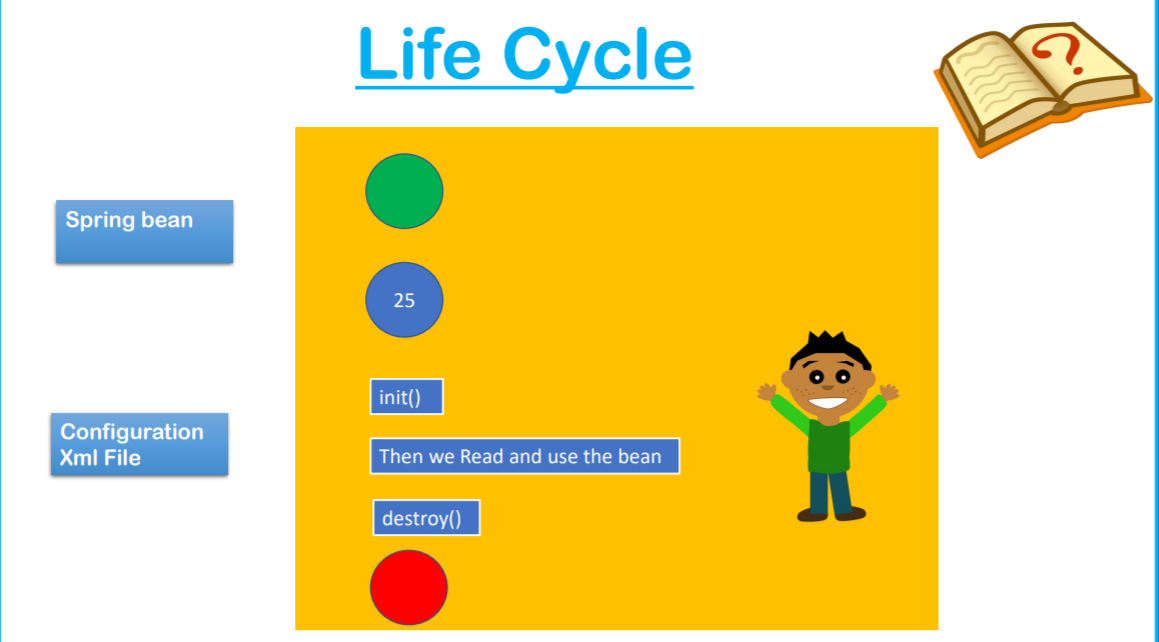
Person [name=Suman, age=18, certi=Certi [name=JAVA-OCJB], listOfCourses=[Python, JAVA, JS]]

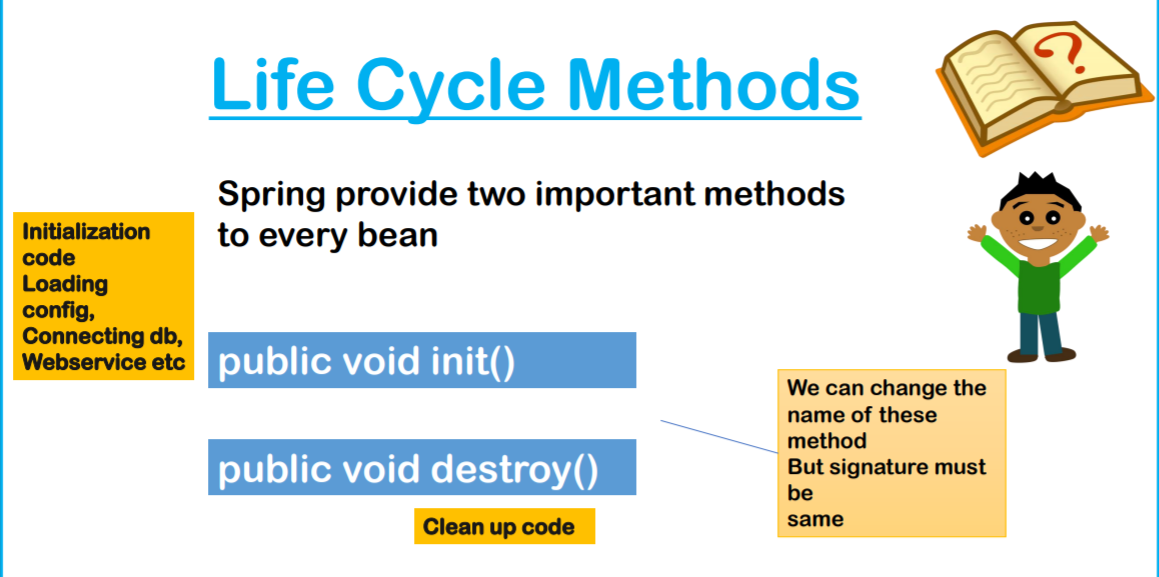
a = 60 | b = 40

SUM =100

##Lecture 13,14

Life Cycle of Beans





When Spring makes an Object , We will study about the different stages of the object.

Spring provides 2 important methods for any bean.

public void init()

public void destroy()

We can change the name of the above default methods but we cannot hinder the signature.

init()

Initialisation code

* load configuration
* config db
* connecting db
* webservice etc

destroy()

clean up code

* close the connection
* flushing

We will declare

Spring Bean (class) and configuration file(.xml metadata)

get()

Instantiation of Objects(state ininitialised)

**\*\*\* init() is called after state initialisation**

{

all the prerequisite has been done

}

then we read and use the bean

destroy() called

{

all the clean-up code

}

Object deallocated by GC

*CONFIGURATION TECHNIQUE*

* XML
* Java Annotation
* Spring Interface (Overriding few method provide by Spring)

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

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

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

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

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

http://www.springframework.org/schema/context/spring-context.xsd

">

*<!-- <context:annotation-config /> -->*

*<!-- enables all the annotations in config.xml -->*

*<!-- enables only @PostConstruct,@PreDestroy -->*

<bean class="org.springframework.context.annotation.CommonAnnotationBeanPostProcessor"/>

<bean class="com.spring.core.lifecycle.Samosa" name="s1"

init-method="hey" destroy-method="destroy">

<property name="price" value="10"/>

</bean>

<bean class="com.spring.core.lifecycle.CocaCola" name="c1" p:price="15"/>

<bean class="com.spring.core.lifecycle.BreadPakoda" name="b1" p:price="20"/>

</beans>

**Samosa.java**

**package** com.spring.core.lifecycle;

/\*lifecycle methods using xml

\*/

**public** **class** Samosa {

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

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

**return** price;

}

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

System.*out*.println("Setting price of Samosa via setter");

**this**.price = price;

}

**public** Samosa() {

**super**();

// **TODO** Auto-generated constructor stub

}

@Override

**public** String toString() {

**return** "Samosa [price=" + price + "]";

}

**public** **void** hey() {

System.*out*.println("INSIDE -> renamed hey :p init() ");

}

**public** **void** destroy() {

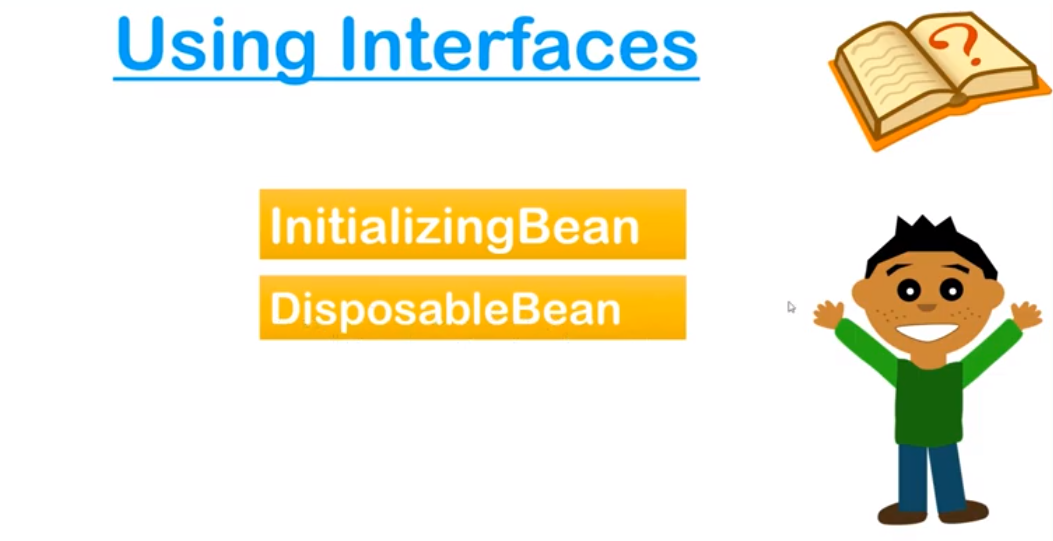
System.*out*.println("INSIDE -> destroy() ");

}

}

##Lecture 15

via Interface



CocaCola.java

**package** com.spring.core.lifecycle;

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

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

/\*lifecycle methods using interface

\*/**public** **class** CocaCola **implements** InitializingBean,DisposableBean{

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

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

**return** price;

}

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

System.*out*.println("Seeting price of coke via setters");

**this**.price = price;

}

**public** CocaCola() {

**super**();

// **TODO** Auto-generated constructor stub

}

@Override

**public** String toString() {

**return** "CocaCola [price=" + price + "]";

}

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

// **TODO** Auto-generated method stub

//init code here

System.*out*.println("taking Coke: init via interface");

}

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

// **TODO** Auto-generated method stub

//destroy code here

System.*out*.println("returning Coke bottle: destroy via interface");

}

}

##Lecture 16

via Annotations



**BreadPakoda.java**

**package** com.spring.core.lifecycle;

//lifecycle using @nnotation

**import** javax.annotation.PostConstruct;

**import** javax.annotation.PreDestroy;

**public** **class** BreadPakoda {

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

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

**return** price;

}

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

**this**.price = price;

System.*out*.println("Setting price of BreadPakoda via setter");

}

@Override

**public** String toString() {

**return** "BreadPakoda [price=" + price + "]";

}

**public** BreadPakoda() {

**super**();

// **TODO** Auto-generated constructor stub

}

//for 9+ Java have to mention the dependencies

@PostConstruct

**public** **void** start() {

System.*out*.println("INIT methods using @nnotations");

}

@PreDestroy

**public** **void** end(){

System.*out*.println("DESTROY methods using @nnotations");

}

}

## 1.  Introduction

Spring allows us to attach custom actions to [bean creation and destruction](https://www.baeldung.com/running-setup-logic-on-startup-in-spring). We can, for example, do it by implementing the InitializingBean and DisposableBean interfaces.

In this short tutorial, we'll look at a second possibility: the @PostConstruct and @PreDestroy annotations.

## 2.  @PostConstruct

Spring calls methods annotated with **@PostConstruct** only once, just after the initialization of bean properties. Keep in mind that these methods will run even if there is nothing to initialize.

The method annotated with @PostConstruct can have any access level but it can't be static.

One example usage of @PostConstruct is populating a database. During development, for instance, we might want to create some default users:

@Component

public class DbInit {

@Autowired

private UserRepository userRepository;

@PostConstruct

private void postConstruct() {

User admin = new User("admin", "admin password");

User normalUser = new User("user", "user password");

userRepository.save(admin, normalUser);

}

}

ove example will first initialize UserRepository and then run @PostConstruct method.

## 3. @PreDestroy

A method annotated with @PreDestroy runs only once, just before Spring removes our bean from the application context.

Same as with @PostConstruct, the methods annotated with @PreDestroy can have any access level but can't be static.

@Component

public class UserRepository {

private DbConnection dbConnection;

@PreDestroy

public void preDestroy() {

dbConnection.close();

}

}

The purpose of this method should be to release resources or perform any other clean-up tasks before the bean gets destroyed, for example closing a database connection.

## 4. Java 9+

## Note that both @PostConstruct and @PreDestroy annotations are part of Java EE. And since [Java EE has been deprecated in Java 9](https://www.baeldung.com/java-enterprise-evolution) and removed in Java 11 we have to add an additional dependency to use these annotations:

*<!-- dependecy to use annotation -->*

*<!-- @PostConstruct,@PreDestroy -->*

<dependency>

<groupId>javax.annotation</groupId>

<artifactId>javax.annotation-api</artifactId>

<version>1.3.2</version>

</dependency>

<?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:context="http://www.springframework.org/schema/context"

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

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

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

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

http://www.springframework.org/schema/context/spring-context.xsd

">

*<!-- enables all the annotations when declared in config.xml -->*

<context:annotation-config />

*if we want to enable specifically those two only we will make <bean /> of :*

*ctrl+shift+t*

search -> CommonAnnotationBeanPostProcessor

*<!-- enables only @PostConstruct,@PreDestroy when declared in config.xml -->*

<bean class="org.springframework.context.annotation.CommonAnnotationBeanPostProcessor"/>

**TestLifecycle.java**

***package*** com.spring.core.lifecycle;

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

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

**public** **class** TestLifecycle {

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

// **TODO** Auto-generated method stub

AbstractApplicationContext context = **new** ClassPathXmlApplicationContext("com/spring/core/lifecycle/config.xml");

// to invoke destroy() we have to enable pre-shutdown.

context.registerShutdownHook();

/\* Registering shutdown hook

\* when you have to destroy object call the hooked-> destroy method()

\*/

Samosa s1 = (Samosa)context.getBean("s1");

System.*out*.println(s1);

System.*out*.println("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_");

CocaCola c1 = (CocaCola) context.getBean("c1");

System.*out*.println(c1);

/\*

\* the order of many beans may differ but individual order of every beans

\* setting->init->destroy

\*/

System.*out*.println("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_");

BreadPakoda b1 = (BreadPakoda) context.getBean("b1");

System.*out*.println(b1);

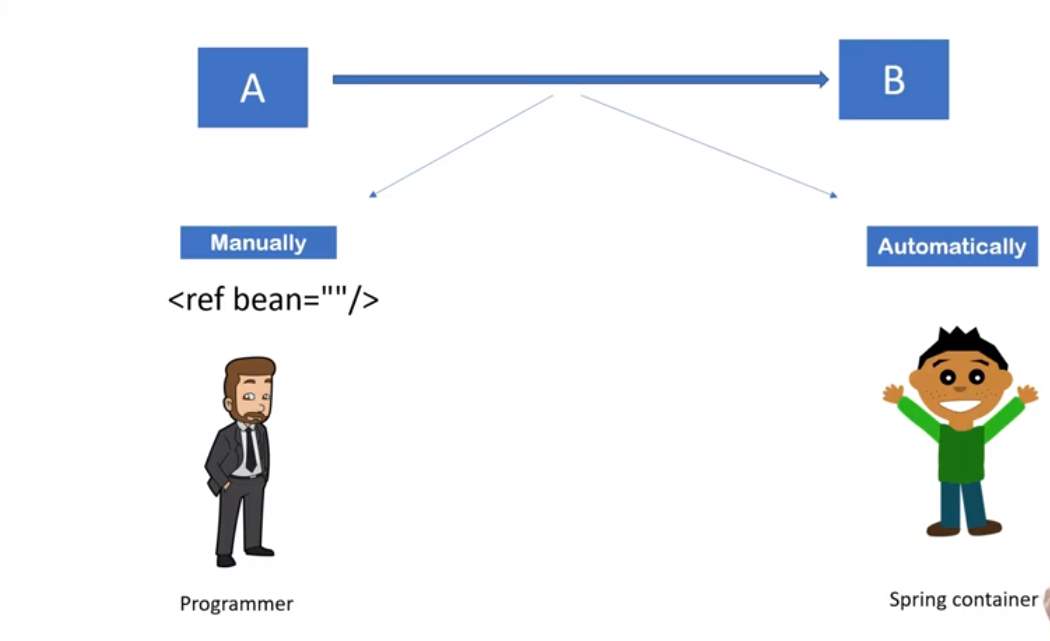
}

}

**##Lecture 17,18**

**AUTO-WIRING in Spring**

1. Its a feature of Spring in which Spring Container injects the dependencies automatically.
2. Its used to inject reference(OBJECT) type only so cannot be used for primitive type. And its good because we mostly use reference as a medium of sending values.
3. B -> A B is a dependency of A ( Spring container will automatically link this)



Auto wiring can be achieved by:

1. XML

*modes*

* 1. no (By default disabled)
  2. byName
  3. byType
  4. constructor
  5. autodetect (deprecated since Spring 3)

1. Annotations
   1. @Autowired

|  |  |
| --- | --- |
| **Auto-Wiring** | |
| **Advantages** | **Disadvantages** |
| * Automation on DI | * No control of programmer |
| * Less code | * Can't be used for primitive and String values |

using XML

byName

name of ref variable and name of dependency bean should be exact same

does not throw exception if name mismatch simply it will not inject

the ref will become null.

byType

checks type, no need for same-name

but it can confuse if two type of bean of same ref type and can lead to Exception (expected single matching but found :2..)

\*BOTH IS USING SETTER INJECTION BY DEFAULT

autowire="byType"

autowire="byName"

autowire="constructor"

<!-- when using byType ensure only 1 type matches -->

autowire="constructor"

If only one type of beam matches with parameter ,it will auto-wire irrespective of name.

If more than one bean type matches then there should be at least one with identical name as of constructor parameter, else it will not inject(ref=null)

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

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

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

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

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

http://www.springframework.org/schema/context/spring-context.xsd

">

<bean class="com.spring.core.auto\_wire.Address"

name="address1"

p:street="Kalikapoor"

p:city="Delhi"

p:pin="700107"/>

<bean class="com.spring.core.auto\_wire.Address"

name="address"

p:street="Doranda"

p:city="Ranchi"

p:pin="800107"/>

*<!-- we will use auto-wiring here -->*

*<!-- 1) it will search for name "address" in existing bean -->*

*<!-- so it need to be taken care while matching -->*

<bean class="com.spring.core.auto\_wire.Worker" name="w1"

autowire="byName"

/>

*<!-- autowire="byType" -->*

*<!-- autowire="byName" -->*

*<!-- autowire="constructor" -->*

*<!-- when using byType ensure only 1 type matches -->*

</beans>

Worker.java

**package** com.spring.core.auto\_wire;

**public** **class** Worker {

**private** Address address;

**public** Address getAddress() {

**return** address;

}

**public** **void** setAddress(Address address) {

System.*out*.println("Setters are used for injection");

**this**.address = address;

}

@Override

**public** String toString() {

**return** "Worker [address=" + address + "]";

}

**public** Worker() {

**super**();

// **TODO** Auto-generated constructor stub

}

**public** Worker(Address address3) {

**super**();

System.*out*.println("Parameterized constructor is used for injection");

**this**.address = address3;

}

}

Address.java

**package** com.spring.core.auto\_wire;

**public** **class** Address {

**private** String street;

**private** **int** pin;

**private** String city;

@Override

**public** String toString() {

**return** "Address [street=" + street + ", pin=" + pin + ", city=" + city + "]";

}

**public** String getStreet() {

**return** street;

}

**public** **void** setStreet(String street) {

**this**.street = street;

}

**public** **int** getPin() {

**return** pin;

}

**public** **void** setPin(**int** pin) {

**this**.pin = pin;

}

**public** String getCity() {

**return** city;

}

**public** **void** setCity(String city) {

**this**.city = city;

}

}

**TestAutoWiring.java**

**package** com.spring.core.auto\_wire;

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

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

**public** **class** TestAutoWiring {

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

// **TODO** Auto-generated method stub

AbstractApplicationContext context = **new** ClassPathXmlApplicationContext("com/spring/core/auto\_wire/autoconfig.xml");

// to skip type-casting

Worker w1 = context.getBean("w1",Worker.**class**);

System.*out*.println(w1);

}

}

##Lecture 19,20

By Annotations

1) By using annnotation @Autowired on property (Automatically by Spring)

2) By using annnotation @Autowired on setter

3) By using annnotation @Autowired on constructor

*\*\*If one dependency to be inject , name doesn’t matter*

@Qualifier

used for specific injection via name of bean passing to @Qualifier() if more than one bean found.

@Autowired

@Qualifier("permanent")

positon of @Autowired

@Qualifier can be interchanged

Injects “permanent” bean if declared

if not-”No such bean found Exception”

valid on setter and property

but not on constructors

if present at both setter and constructor

setter will get priority.

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

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

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

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

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

http://www.springframework.org/schema/context/spring-context.xsd

">

<context:annotation-config/>

*<!-- recommended to set same bean name as that name for variable -->*

<bean class="com.spring.core.auto\_wire.annotations.Address"

name="current"

p:street="Kalikapoor"

p:city="Delhi"

p:pin="700107"/>

<bean class="com.spring.core.auto\_wire.annotations.Address"

name="permanent"

p:street="White Field"

p:city="Bangalore"

p:pin="701605"/>

<bean class="com.spring.core.auto\_wire.annotations.Worker" name="w1"/> </beans>

**Address.java**

**package** com.spring.core.auto\_wire.annotations;

**public** **class** Address {

**private** String street;

**private** **int** pin;

**private** String city;

@Override

**public** String toString() {

**return** "Address [street=" + street + ", pin=" + pin + ", city=" + city + "]";

}

**public** String getStreet() {

**return** street;

}

**public** **void** setStreet(String street) {

**this**.street = street;

}

**public** **int** getPin() {

**return** pin;

}

**public** **void** setPin(**int** pin) {

**this**.pin = pin;

}

**public** String getCity() {

**return** city;

}

**public** **void** setCity(String city) {

**this**.city = city;

}

}

**Worker.java**

**package** com.spring.core.auto\_wire.annotations;

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

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

**public** **class** Worker {

// @Autowired

@Qualifier("current")

@Autowired

**private** Address address;

**public** Address getAddress() {

**return** address;

}

// @Autowired

// @Qualifier("permanent")//setter gets priority over properties

**public** **void** setAddress(Address address) {

System.*out*.println("Setters are used for injection");

**this**.address = address;

}

@Override

**public** String toString() {

**return** "Worker [address=" + address + "]";

}

**public** Worker() {

**super**();

// **TODO** Auto-generated constructor stub

}

// @Autowired

**public** Worker(Address address3) {

**super**();

System.*out*.println("Parameterized constructor is used for injection");

**this**.address = address3;

}}

**TestAutoWiring.java**

package com.spring.core.auto\_wire.annotations;

import org.springframework.context.support.AbstractApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class TestAutoWiring {

public static void main(String[] args) {

// TODO Auto-generated method stub

AbstractApplicationContext context = new ClassPathXmlApplicationContext("com/spring/core/auto\_wire/annotations/autoconfig.xml");

// to skip type-casting

Worker w1 = context.getBean("w1",Worker.class);

System.out.println(w1);

}

}

##Lecture 21

Spring Standalone Collections

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

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

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

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

*<!-- this is our 1st bean and we are injecting collections -->*

<bean name="emp1" class="com.spring.core.collections.Employees">

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

<property name="phones">

<list>

<value>9563265148</value>

<value>9563265148</value>

<value>9563265148</value>

<null/>

</list>

</property>

***<!-- we cannot use this list outside this bean because its not a standalone list-->***

<property name="pin">

<value>828130</value>

</property>

<property name="tags">

<list>

</list>

</property>

<property name="address">

<set>

<value>Delhi</value>

<value>Lucknow</value>

<value>Kanpur</value>

<value>Delhi</value>

</set>

</property>

<property name="course">

<map>

<entry key="Java" value="3"/>

<entry key="Python" value="4"/>

<entry key="JS" value="6.5"/>

</map>

</property>

<property name="general">

<props>

<prop key="gender">male</prop>

<prop key="age">24</prop>

<prop key="passportAvailbility">YES</prop>

</props>

</property>

</bean>

</beans>

* We will use standalone collection to achieve globality.
* We can also add specificity to the collection.

<list> </list> -> automatically impletation class has been taken (unaware its a linked list / array-list...)

By default its ArrayList

same for set <set></set> (unaware its a linked hash set / hash set...)

Stand-alone Collections can be used for saving database credentials in a collection inside a file to avoid recompilation every time.

alone\_config.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:context="http://www.springframework.org/schema/context"

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

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

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

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

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

http://www.springframework.org/schema/context/spring-context.xsd

http://www.springframework.org/schema/util

http://www.springframework.org/schema/util/spring-util.xsd

">

*<!-- my list stand-alone custom list -->*

*<!-- list-class="java.util.Vector" -->*

<util:list

list-class="java.util.LinkedList"

id="mybestfriends">

<value>Suman</value>

<value>Satyam</value>

<value>Sukanya</value>

<value>Apurv</value>

<value>Ashutosh</value>

<value>Nitish</value>

<value>Soumadeep</value>

<value>null</value>

</util:list>

*<!-- this is our 1st bean -->*

<util:map map-class="java.util.TreeMap" id="fees">

*<!-- TreeMap is used for key wise sorting -->*

*<!-- <util:map map-class="java.util.HashMap" id="fees"> -->*

<entry key="Spring Core" value="8000"/>

<entry key="Django" value="6000"/>

<entry key="React" value="5000"/>

</util:map>

*<!-- Properties is a class extending Hashtable -->*

<util:properties id="props">

<prop key="college">HIT</prop>

<prop key="branch">Chemical</prop>

<prop key="since">2015</prop>

</util:properties>

*<!-- this is our 1st bean for Person -->*

<bean class="com.spring.core.standalone\_collections.Person"

name="p1">

<property name="friends">

<ref bean="mybestfriends"/>

</property>

*<!-- using self closing property tag & reusing the Linkedlist-->*

<property name="feeStructure" ref="fees"/>

<property name="props" ref="props"/>

</bean>

<bean class="com.spring.core.standalone\_collections.Person"

name="p2">

<property name="friends" ref="mybestfriends"/>

<property name="props" ref="props"/>

<property name="feeStructure" ref="fees"/>

</bean>

</beans>

Person.java

**package** com.spring.core.standalone\_collections;

**import** java.util.List;

**import** java.util.Map;

**import** java.util.Properties;

**public** **class** Person {

**private** List<String> friends;

**private** Map<String,Integer> feeStructure;

**private** Properties props;

**public** Properties getProps() {

**return** props;

}

**public** **void** setProps(Properties props) {

**this**.props = props;

}

**public** Map<String, Integer> getFeeStructure() {

**return** feeStructure;

}

**public** **void** setFeeStructure(Map<String, Integer> feeStructure) {

**this**.feeStructure = feeStructure;

}

**public** List<String> getFriends() {

**return** friends;

}

@Override

**public** String toString() {

**return** "Person [friends=" + friends + ",\nfeeStructure=" + feeStructure + "]\n"+"Properties ->"+props+"\n";

}

// we will use setter injection

**public** **void** setFriends(List<String> friends) {

**this**.friends = friends;

}

}

TestStanaloneCollections.java

**package com.spring.core.standalone\_collections;**

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

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

**public** **class** TestStanaloneCollections {

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

// **TODO** Auto-generated method stub

ApplicationContext context =

**new** ClassPathXmlApplicationContext("com/spring/core/standalone\_collections/alone\_config.xml");

Person p1 = context.getBean("p1",Person.**class**);

System.*out*.println(p1);

System.*out*.println(p1.getFriends().getClass());

System.*out*.println(p1.getFriends().getClass().getName());

System.*out*.println(p1.getFeeStructure().getClass());

Person p2 = context.getBean("p2",Person.**class**);

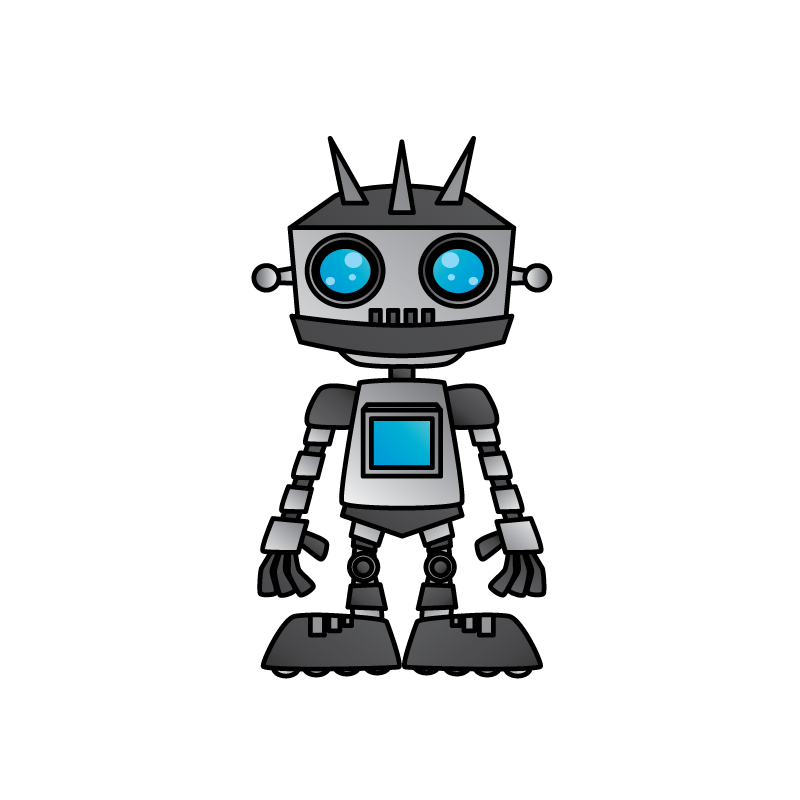
System.*out*.println(p2);

System.*out*.println(p2.getFeeStructure().getClass());

// if re isNull dont invoke getClass()

}

}



**\*\* ##Lecture 22,23,24**

**Stereotype Annotations**



* To tell Spring container to make object we use to add the details inside <bean /> in XML
* But we can use few annotations w/o using <bean /> in XML for the same.

**@Component** can be used to tell the Spring Container that this is the class which needs to instantiated.

**<bean />** [**≈**](https://math.stackexchange.com/questions/864606/difference-between-≈-≃-and-≅) **@Component**

**ie:**

package sprinhg.core.Student

@Component

Class Student{

}

Student student = new *Student*(); /*/ follows camelCase for automatic standard variable name.*

**Prerequisite:**

We have to tell in XML configuration file the package containing this class.

<context:component-scan base-package=””/>

Scans packages and all the sub packages for the target Class.

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

**@Component**

*it’s stored in stereotype sub-package so comes its name, it creates bean following standard camelCase for class name.*

**@Component(“s1”)**

*it doesn’t creates bean following standard camelCase for class name , instead it creates bean with our custom name “s1”*

**@Value(“”)**

*it’s used to inject values to the instance variables and is declared above instance variables.*

**@Value(“#{idOfStandaloneCollection}”) for collections**

*it’s used to inject stand-alone collections id ref declared in .XML to collection instance variable.*

**Spring Bean Scope**

predefined scopes:

1. Singleton
2. prototype
3. request
4. session
5. globalsession

* \*\*web app specific (request)-> HttpRequest ; session -> HtpSession
* \*\*globalsession -> used for portlet application

Singleton (It gets applied by default) so, if you haven't configured any scope.

It provides same reference of the 1st created object to all the subsequent objects, made of the same class. We can verify this by hashCode()

Prototype will create a brand new object every time an instantiation is demanded.

Configuew Bean Scope

via XML

<bean class=”” name=”” scope=””/>

via Annotations

@Component("s1")

@Scope("prototype")

**stereo\_config.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:context="http://www.springframework.org/schema/context"

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

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

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

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

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

http://www.springframework.org/schema/context/spring-context.xsd

http://www.springframework.org/schema/util

http://www.springframework.org/schema/util/spring-util.xsd

">

<context:component-scan base-package="com.spring.core.stereotype"></context:component-scan>

<util:list list-class="java.util.Vector"

id="course">

<value>Spring</value>

<value>Git</value>

<value>Django</value>

<value>React</value>

</util:list>

<bean class="com.spring.core.stereotype.Teacher"

name="teacher" scope="prototype"/>

*</beans>*

**Student.java**

**package** com.spring.core.stereotype;

**import** java.util.List;

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

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

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

@Component("s1")

@Scope("prototype")

**public** **class** Student {

@Value("Suman Shekhar")

**private** String studentName;

@Value("Bangalore")

**private** String city;

@Value("#{course}")

**private** List<String> courses;

**public** String getStudentName() {

**return** studentName;

}

**public** **void** setStudentName(String studentName) {

**this**.studentName = studentName;

}

@Override

**public** String toString() {

**return** "Student [studentName=" + studentName + ", city=" + city + ", courses=" + courses + "]";

}

**public** String getCity() {

**return** city;

}

**public** List<String> getCourses() {

**return** courses;

}

**public** **void** setCourses(List<String> courses) {

**this**.courses = courses;

}

**public** **void** setCity(String city) {

**this**.city = city;

}

}

**Teacher.java**

***package*** *com.spring.core.stereotype;*

**public** **class** Teacher {

}

**TestStereo.java**

**package** com.spring.core.stereotype;

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

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

**import** com.spring.core.standalone\_collections.Person;

**public** **class** TestStereo {

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

// **TODO** Auto-generated method stub

ApplicationContext context =

**new** ClassPathXmlApplicationContext("com/spring/core/stereotype/stereo\_config.xml");

Student s1 = context.getBean("s1",Student.**class**);

System.*out*.println(s1);

System.*out*.println(s1.getCourses().getClass());

System.*out*.println(s1.hashCode());

System.*out*.println("\n\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\n");

Student s2 = context.getBean("s1",Student.**class**);

System.*out*.println(s2);

System.*out*.println(s2.hashCode());

Teacher t1 = context.getBean("teacher",Teacher.**class**);

Teacher t2 = context.getBean("teacher",Teacher.**class**);

System.*out*.println("\nTeacher prototype Scope \n\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_");

System.*out*.println(t1.hashCode());

System.*out*.println(t2.hashCode());

}

}

**##Lecture 25,26,27**

**SpEL**

Spring Expression Language

@Component("s1")

@Scope("prototype")

**public** **class** Student {

@Value("Suman Shekhar")

**private** String studentName;

@Value("Bangalore")

**private** String city;

@Value("#{course}")

**private** List<String> courses;

At runtime this List will get resolved -> #{course}

The above is an example of SpEL

SpEL supports

1. Parsing
2. Executing

The above two is done with the help of:

@Value(**“#{expression}”**)

**“expression” can be**

* Classes, variable, methods, static methods , constructors & Objects
* primitive values .. @value(“#{ 3+5 }”)
* Symbols
  + char
  + alphanumeric
  + operators i.e. ternary .. @value(“#{ 8>6? 88:55 }”)
  + keywords
  + special symbols which return a value

Spring provides API of SpEL using class *SpelExpressionParser*

*SpelExpressionParser* spell = **new** SpelExpressionParser();

//Don’t use #{} when using SpelExpressionParser class

Expression expression = spell.parseExpression("0>=1 ? true:false");

System.*out*.println(expression.getValue());

*Using SpEL for objects and static contents*

1. *T(class).staticmethod(param)*

@Value("#{ T(java.lang.Math).sqrt(169) }")

//SpEL used for static method call

**private** **double** y;

1. *T(class).staticvariable*

@Value("#{ T(java.lang.Math).E }")

//SpEL used for static variables access

**private** **double** e;

1. *new ClassName()*

@Value("#{new java.lang.String('SpEL')}")

//SpEL used object creation

**private** String topic;

1. boolean condition

@Value("#{10<=11 ? true:false}")

//SpEL used for handling boolean

**private** **boolean** isActive;

**spel\_config.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:context="http://www.springframework.org/schema/context"

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

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

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

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

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

http://www.springframework.org/schema/context/spring-context.xsd

http://www.springframework.org/schema/util

http://www.springframework.org/schema/util/spring-util.xsd

">

<context:component-scan base-package="com.spring.core.SpEL"/>

</beans>

**Demo.java**

***package*** com.spring.core.SpEL;

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

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

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

@Component

@Scope("prototype")

**public** **class** Demo {

@Value("31")

**private** **int** w;

@Value("#{30+39}")//SpEL

**private** **int** x;

@Value("#{ T(java.lang.Math).sqrt(169) }")

//SpEL used for static method call

**private** **double** y;

@Value("#{ T(java.lang.Math).E }")

//SpEL used for static variables access

**private** **double** e;

@Value("#{new java.lang.String('SpEL')}")

//SpEL used for Object creation

**private** String topic;

@Value("#{10<=11 ? true:false}")

//SpEL used for handling boolean

**private** **boolean** isActive;

**public** **boolean** isActive() {

**return** isActive;

}

@Override

**public** String toString() {

**return** "Demo [w=" + w + ", x=" + x + ", y=" + y + ", e=" + e + ", topic=" + topic + ", isActive=" + isActive

+ "]";

}

**public** **void** setActive(**boolean** isActive) {

**this**.isActive = isActive;

}

**public** String getTopic() {

**return** topic;

}

**public** **void** setTopic(String topic) {

**this**.topic = topic;

}

**public** **int** getW() {

**return** w;

}

**public** **void** setW(**int** w) {

**this**.w = w;

}

**public** **int** getX() {

**return** x;

}

**public** **void** setX(**int** x) {

**this**.x = x;

}

**public** **double** getY() {

**return** y;

}

**public** **void** setY(**double** y) {

**this**.y = y;

}

**public** **double** getE() {

**return** e;

}

**public** **void** setE(**double** e) {

**this**.e = e;

}

}

**TestSpell.java**

***package*** com.spring.core.SpEL;

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

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

**import** org.springframework.expression.Expression;

**import** org.springframework.expression.spel.standard.SpelExpressionParser;

**import** com.spring.core.stereotype.Student;

**public** **class** TestSpell {

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

// **TODO** Auto-generated method stub

ApplicationContext context =

**new** ClassPathXmlApplicationContext("com/spring/core/SpEL/spel\_config.xml");

Demo d1 = context.getBean("demo",Demo.**class**);

System.*out*.println(d1);

SpelExpressionParser spell = **new** SpelExpressionParser();

// No #{} when using SpelExpressionParser class

Expression expression = spell.parseExpression("0>=1 ? true:false");

System.*out*.println(expression.getValue());

}

}

*ctrl+shift+e* -> Class Browser

**##Lecture 28**

**Completely Removing XML:**

**TestConfigFree.java**

**package** com.spring.core.javaconfig;

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

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

**public** **class** TestConfigFree {

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

// **TODO** Auto-generated method stub

/\*

\*\*\* using xml

\* ApplicationContext context = new

\* ClassPathXmlApplicationContext("com/spring/core/javaconfig/javaconfig.xml");

\* Student s1 = context.getBean("student",Student.class);

\* System.out.println(s1);

\*/

ApplicationContext context = **new** AnnotationConfigApplicationContext(JavaConfig.**class**);

Student s1 = context.getBean("stud",Student.**class**);

/\* we can user either the array elements or the default method-name ***getStudent*** not both \*/

System.*out*.println(s1);

s1.study();

}

}

**Samosa.java**

**package** com.spring.core.javaconfig;

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

//@Component

**public** **class** Samosa {

**public** **void** order() {

System.*out*.println("Samosa ordered !!");

}

}

**Student.java**

**package** com.spring.core.javaconfig;

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

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

//@Component

**public** **class** Student {

@Autowired//automatic injection

**private** Samosa units;

**public** **void** study() {

**this**.units.order();

System.*out*.println("STYDYING: \"\"The Pen in mightier than sword");

}

**public** Samosa getUnits() {

**return** units;

}

// through constructor injection

// public Student(Samosa units) {

// super();

// this.units = units;

// }

**public** **void** setUnits(Samosa units) {

**this**.units = units;

}

}

JavaConfig.java

**package** com.spring.core.javaconfig;

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

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

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

/\*this class is handling configuration stuff

\* So we have to inform this a maker to tell

\* the Spring Container via @ComponentScan

\* that its a config file

\* scanning using

\* -> @ComponentScan(basePackages = "com.spring.core.javaconfig")

\* \*/

@Configuration

//No need to use @ComponentScan when using @Bean

//@ComponentScan(basePackages = "com.spring.core.javaconfig")

**public** **class** JavaConfig {

// using w/o declaring @Component on the class

// @Bean

@Bean(name = {"stud","student","schoolBuoy"})

// through these names we can access the beans,

but default name is that of the method ie ***getStudent***

**public** Student getStudent() {

**return** **new** Student();

}

@Bean

**public** Samosa getSamosa() {

**return** **new** Samosa();

}

}

1. We can inject dependencies using explicit constructor injection or Autowiring
2. ***@Configuration*** is used for defining the configuration class
3. ***@ComponentScan(basePackages = "com.spring.core.javaconfig")***  can be used to search the defile packages and sub packages
4. ***@Component*** is used in conjugation with [//@ComponentScan](mailto://@ComponentScan)
5. ***@Bean*** can be used to avoid [//@ComponentScan](mailto://@ComponentScan) tech