SPRING
The Legend Java

BY
Mr.NATARAJ Sir

NARESH TECHNOLOGY

SPRING

version : 3.x & 4.x

BY NATHRAZ SIR

Written by
P.Rajendra

BASICS

→ C, C++, Java are programming languages.

→ Jdbc, servlet, jsp, ejb, jms, java mail, jta, jpa, etc.. java technologies

→ Struts, JSF, spring, Hibernate etc.. Java frameworks.

Programming languages

→ It is directly installable s/w having basic features to develop s/w App's.

→ It defines syntaxes (rules), semantics (structure) of programming.

→ These are base to create s/w technologies, frameworks, tools, DB s/w, operating system

→ These are raw materials of s/w programming.

Exp : C, C++, Java

s/w technology

→ It is a s/w specification having set of rules & guidelines to develop s/w by using programming language support.

→ s/w technology is not installable but s/w created based on s/w technology is installable. Working with s/w is nothing but working with s/w technology.

→ In Java environment, technologies rules are nothing but interfaces & guidelines are nothing but classes. Abstract class represent both rules & guidelines.

Exp: Jdbc, jsp, servlet, EJB etc..

→ Jdbc technology gives rules & guidelines to develop driver using java language Working with jdbc driver is nothing but working with jdbc technologies.

→ They are 2 types of s/w technologies

a) Open technologies: (Here rules & guidelines of Technology are open, all s/w vendor companies to develop the s/w).

Exp: Jdbc, servlet, JSP etc..

b) Proprietary Technologies

(Here the vendor company who has given technologies is only allowed to develop the s/w's based on the technologies)

Exp: All microsoft technologies like ASP.net,....

Frame Works

→ Framework is installable sw that uses existing technologies internally to simplify the application development having ability to generate common logics of the appl^n externally.

→ Framework is installable sw that provides abstraction layer on existing technologies to simplify the application development process.

Exp: struts, hibernate, spring etc...

→ While working with technologies the programmer has to develop both common logics & app specific logics of the Application.

→ While working with framework the programmer has to develop only specific logics because the common logics will be generated by the framework dynamically.

→ Every framework uses technology internally but it never make program to know about it. This is nothing but providing abstraction layer (hidding implementations / Internals).

Plain Jdbc Application

→ Register Jdbc driver (load driver class) } (common logics)
→ Establish the connection
→ Create stmt object
→ Send & execute sql Query in DB sw } (specific logic)
→ Gather results & process results
→ close Jdbc object } (common logic)

Spring Jdbc Application

→ Get JdbcTemplate object.
→ Send & execute SQL Query in DB sw } (specific logics)
→ Gather results & process results

→ While working with technology we need to write same common logics in every application in that technology this is called "Boiler plate code problem"

→ While working with framework we just need to develop application specific logics that means it solves "boiler plate code problem".

* Based on the kind of Application logics we develop, there are 3 types of java framework

a) Web App Framework

→ Provides abstraction layer on servlet, jsp technology.
→ simplies MVC architecture based web app development.

JSF → from sunMS (oracle corp) (2)
struts → from Appache
Webwork → from open symphony
ADF → from oracle corp (3)
Spring MVC → from Interface21 (1)

b) ORM frameworks

→ Provides Abstraction layer on jdbc technology
→ simplifies object based o-r mapping persistence logic development
→ Allows to portal / DB Independent Persistence logic.

Hibernate → from softtree (RedHat)
Toplink → from oracle corp
eJB → from Appache
iBatis → from Apache
JDO → from Adobe

c) APPln Frame Work

→ JEE provides abstraction layer on multiple technologies like jdbc, servlet, jsp, jndi, jms, jta, ejb, rmi etc..
→ Allows to develop all kinds of logics like business logic, persistence logic, presentation logic, Integration logic etc..
→ Allows to develop all kinds of Application like standalone application, web appln's, distributed App's etc..

Exp: spring --> from Interface21

---

* Based on mode of development they allow, there are 2 kinds of frameworks.

a) Invasive Frameworks
b) Non-Invasive Frameworks

a) Invasive framework

⇒ Here the classes of Application should extend from framework "api classes" (or) should implement framework "api interfaces". This process makes application classes as tightly bounded classes of framework.

→ We can't change framework as the application project by changing the libraries (.jar files).

Exp: struts 1.x

b) Non-Invasive Framework

→ Here the classes of appln need not to extend from framework "api" classes or need not to implement framework "api" interface. so our appln classes are not tightly bounded with framework.

→ We can change framework of the appln/ project by changing the libraries (.jar files)

Exp: struts 2.x, spring, hibernate, JSF etc...

i) Applets → for Gaming (1995)

2) Java Beans → for business logic/ service logics development (1996).

Disadvantages :→ cannot apply services, cannot accessed remote clients). Cannot allow to apply for * security, transaction, logging, connection pooling etc... are called services/ middleware services.)

EJB → For business logic/ service logic development (1998).

Advantage :→ Allows both local & remote clients.

→ Allows to apply middleware services.

Disadvantages : Heavy weight & complex.

* EJB is suitable for Banking projects.

Spring (2003) : Initially for business logic/ service logic development with ordinary classes allowing to apply middleware services.

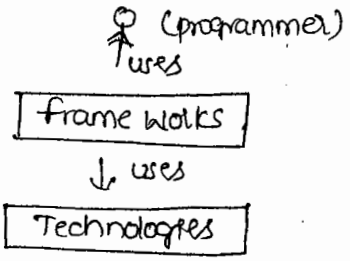
⇒ New spring can be used to develop all kind of logics & Apps in light weight environment.

Before Frameworks : Programmer directly uses technologies to develop both common & specific logics of the application.

11/6/15

With framework

→ Programmer uses frameworks to develop only specific logic because the common logics will generated by frameworks internally by using technologies support.



Q Is the spring replacement for struts?

→ using struts we can develop only web apps & there is a provision in struts to develop & use middleware services/ aspects (securities, logging etc..)

→ using spring we can develop all kinds of apps & all kind of logics including webapps. It is also provides environment to develop & use aspects / middleware services.

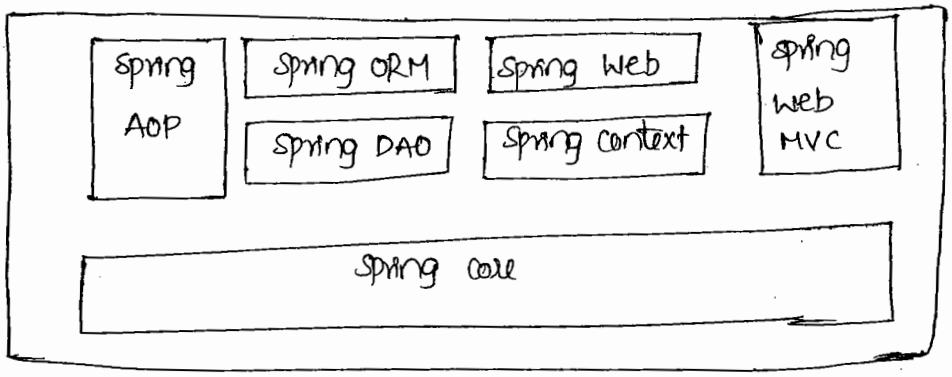
→ Spring is more powerful than struts.

Q Is the spring replacement for JEE/JSE technologies?

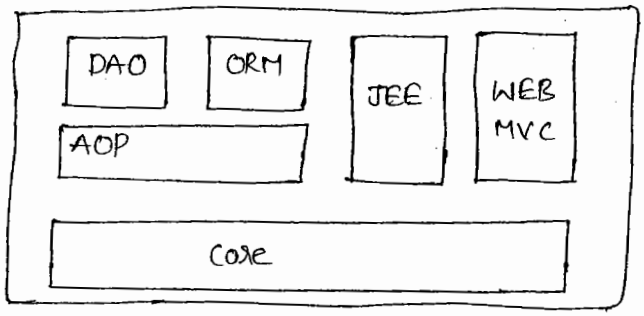
→ NO.

→ Spring complements JEE/JSE technologies by using them internally in the application development process.

Spring 1.x Modules (7) : [Spring is application framework]



Spring 2.x overview Modules (6)



→ Spring 2.x webmodule = spring 1.x web + spring 1.x web MVC Module.

→ Spring 2.x JEE Module = Spring 1.x context Module

Spring Core: This module is base module for other modules provides containers that required to manage various containers that required to manage various spring components resources.

→ useful to develop standalone Applications. Designed for supporting depedency injection (injecting values to objects dynamically).

Spring DAO/JDBC: provides abstraction layer on plain jdbc to simplify the jdbc style persistence logic.

Spring ORM: provides abstraction layer on multiple ORM slw (like hibernate) & simplifies object o-r mapping persistance logic.

Spring Web: provides plugins to make spring apps communicatable from other web framework like struts, JSF and etc.... apps.

Spring Web MVC: Spring own web framework to develop MVC architecture based web App

Spring Context: provides abstraction layer on multiple JSE/JEE module technologies to simplify the development.

a) Mailing App's
b) Distributed App's
c) Messaging App's
d) scheduling App's etc...

Spring AOP : provides new methodology of programming to develop aspect/middleware service logic & to apply them in spring Apps.

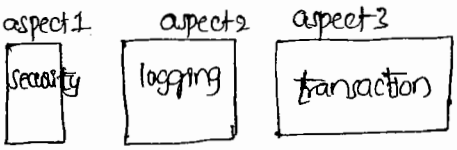
Before AOP :

```
class Bank{
  public void withdraw(_, ___, _)
  {
    security logic
    logging logic
    transaction logic
        Withdraw logic{ bal = bal - amt) }
  }
  public deposite( ) {
    security logic
    logging logic
    transaction logic
    deposite logic{ deposit = bal + amt) }
  }
}
```

Here are writing primary & middleware service logics together

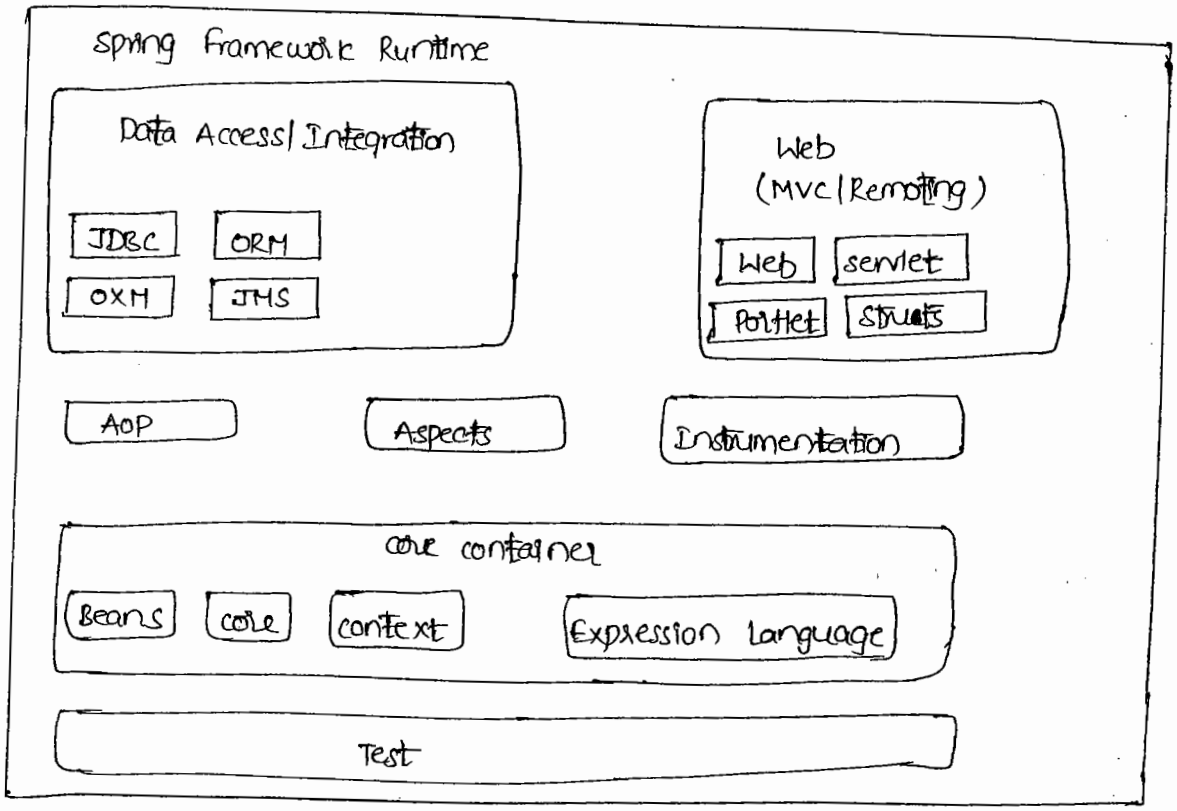
After AOP :

```
class Bank{
  public void withdraw(_,_,_){
    withdraw logic{ ( bal = bal - amt)
                  } (Linked with Aspect through same class)
  public deposite( ) {
    deposite logic{ (deposite = bal + amt)}
  }
```



2/6/15

Spring 3.x overview diagram



→ Spring 3.x is having 20 modules that are grouped into core container, AOP, Instrumentation, Test, Data Access/Integration, MVC/Remoting groups.

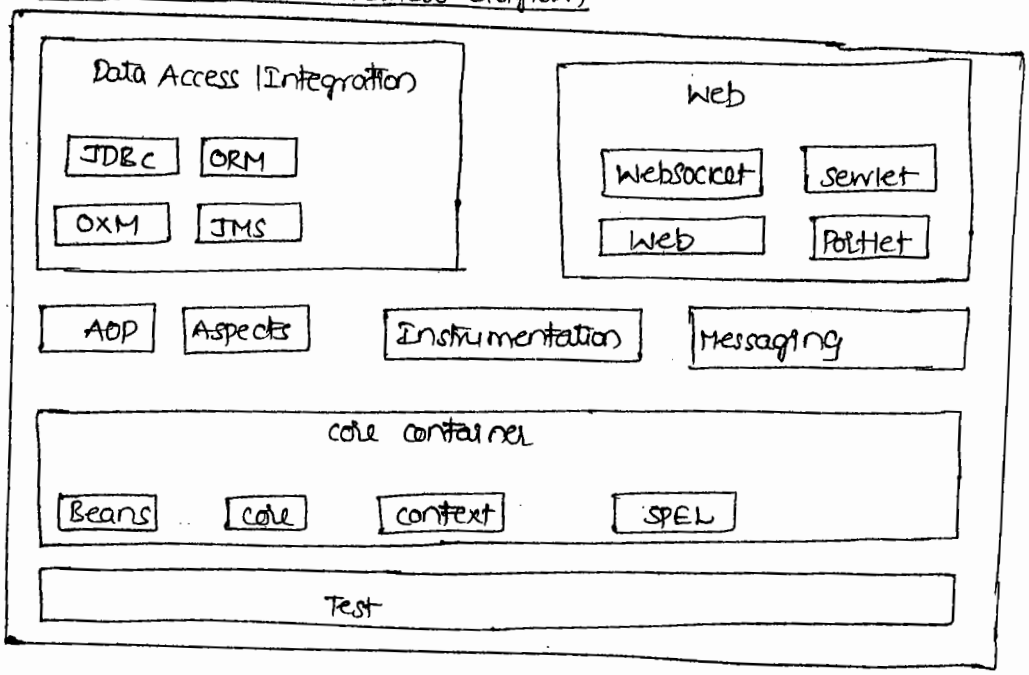
→ Test Module allows to perform unit testing (programmer's testing) on his own piece of code. JUnit does not allow to work with mock objects (dummy objects) where as spring "Test" module allows us to work with mock objects.

→ OXM (Object XML Mapping) is given to convert java objects to XML tags and vice-versa (Alternate to JAXB)

→ Spring 2.x JEE services they placed on Data Access/Integration, MVC/Remoting groups.

→ Portlet: Supports allows to develop "portal" applications multiple windows (webpages) in a webpage.

Spring 4.x framework overview diagram



→ Spring 3.x is compatable with JDK1.5

→ Spring 4.x is compatable with JDK1.6 and supports all jdk1.8 features (in this version all deprecated classes, methods are removed).

→ In spring 4.x messaging is a subframework, in spring that is given alternative to JMS to get messages based communications between components (asynchronous communication between components).

→ Asynchronous communication allows the client to give next request (or) to client side operations with out waiting given request related response from server.

→ WebSocket is a web level protocol that is designed on top of "http" protocol for full two-way communication.

Note Industry uses "MVC" as defacto architecture to develop java based medium/large scale web application.

M → Model → Data + Business Logic + Persistence Logic (like Account Officer)

V → View → Presentation Logic (like Beautician)

C → Controller → Integration Logic (like traffic police/supervisor)

Note Integration logic controlls & monitor all the activities of Application execution. It is responsible to get communication bit view layer & model layer components.

jsp → servlet → Java class | Java Bean → DB s|w
(V) (C) (M)

jsp → servlet → EJB session Bean → EJB Entity Bean → DB s|w
(V) (C) (Business logic) (Persistence Logic)
← Model →

jsp → servlet → EJB session Bean → DAO class (jdbc) → DB s|w
(V) (C) (Business logic) (Persistence logic)
← Model →

struts → EJB session Bean → DAO class (jdbc) → DB s|w
(V)(C) (Business logic) (Persistence logic)
← Model →

→ DAO (Data Access Object): The java class that seperates persistence logic from other logics and makes the persistence logic as reusable logic and flexible logic to modify.

→ Jdbc is light weight but does not allow to develop objects based persistence logic.
→ EJB entity beans heavy weight but allows to develop objects based persistence logic.
↳ To overcome both problems we can use "hibernate".

← Model →
struts → EJB session Bean → Hibernate → DB s|w
(V)(C) (Bussiness logic) (Persistence logic)

← Model →
struts → Webservices → hibernate → DB s|w
(V)(C) (Business logic) (Persistence logic)

← Model →
struts → Spring JEE|Beans → Spring ORM|DAO+AOP → DB s|w
(V)(C) (Business logic) (Persistence logic)

← Model →
SpringMVC → Spring JEE|Beans → Spring ORM|DAO+AOP → DB s|w
(V)(C) (Business logic) (Persistence logic)

Spring MVC → Spring Webservices | Webservices → Spring ORM|DAO+AOP → DB s|w
(V)(C) (Bustesslogic) (Persistence logic)
← Model →

3/6/15

books : spring pro, spring in action

spring

Type : Java Application Framework

Version : 4.x (compatable with jdk 1.6+)
3.x (compatable with jdk 1.5+)

Vendor : Interface 21

Creator : Mr. Rod Jhonson

Spring s/w comes as set of jars

Website : http://spring.io

To download s/w : Download as zip file from spring.io
spring-framework-3.2.RELEASE-with-docs.zip
spring-framework-4.1.6.RELEASE-dist.zip

To install s/w : Extract the zip file

Spring 4.1.6 s/w installation gives
libs → having jar files
docs → api docs, reference docs (pdf)
schemas → xsd file having schema rules.

⇒ Every DTD or XSD file contains rules to construct XML documents
DTD : Document Type Definition
XSD : XML schema Definition

In spring 3.0.0 we have changelog.txt file (to have version information)

→ Spring is an open source, light weight, loosely coupled, aspect oriented, dependency injection based java applications framework to develop various java applications.

⇒ Along with spring f/w installation we get its source code, this makes spring as opensource.

⇒ Opensource s/w are not only free softwares they also expose their source code along with installation.

⇒ Spring is light weight because
a) spring f/w comes as zip file having less size
b) spring gets two light weight containers which can be activated anywhere in the Application.
c) To work with spring container we do not need any application server/ webserver software
d) We can use specific modules of spring without having any link with other modules. For example : We can use spring core module alone or spring core & spring DAO modules.

together with out any link with other modules.

⇒ Spring is loosely coupled because,

→ the degree of dependency is less between the modules.

Exp: TV and remote.

→ In spring, core module is base module for all other modules i.e, we should use every module along with core module, but the remaining modules of spring can be used alone with out having any dependency with other modules of the spring.

Trghtly coupled : Degree of dependency is more between the modules.

Exp keyboard & system.

⇒ Spring is AOP based programming,

+AOP is a new methodology of programming that compliments oop to seperate secondary middle ware services from primary business logic and allows to link them at run time dynamically through some configuration. This gives advantage of reusing middleware service logics and flexibility of enabling/disabling middle ware service logics on primary logics.

middleware services : logging, security, transactions etc..

4/6/15

Dependency Injection (Inversion of control)

1) Dependency lookup : If the resource/class/object is searching and getting its dependent values from other resource then it is dependency lookup.

Here the resource pulls the dependent values.

→ The way we get objects from jndi registry is called "Dependency lookup".

→ The way we get jdbc connection obj from "jdbc con pool" is called "Dependency lookup."

→ If student is getting course material only after putting request for it is called "Dependency lookup".

2) Dependency Injection (Inversion of control):

→ If the underlying server/container/framework/Runtime (like JRE) assigns the value to resource/class/obj dynamically at runtime then it is called "Dependency injection."

Here dependent values will be pushed to the resource.

1) JVM calling constructor automatically to assign initial value to obj when obj is created

2) Servlet Container injects ServletConfig obj to our servlet class obj by calling its init(ServletConfig) method when ServletContainer creates our servlet class obj.

3) student gets course materials the moment he register for course.

What is POJO class, POJI, Java Bean, Bean/component class, Spring Bean.

POJO (plain old java object class)

⇒ The Java class with out any specialities, it is an ordinary java class.

⇒ While developing this class we need not to follow any serious rules.

⇒ The Java class that is not extending from framework api class/technology class and not implementing technology/framework api interfaces is called "POJO class".

Eg1 : class Demo{

}
It is pojo class

Eg2: class Test implements Serializable{

}
POJO class

Eg3: class Test extends Demo{

}
class Demo{

}
Test, Demo are pojo classes

Eg4: class Test extends Thread{

}
Test is a pojo class

Eg5: class Test extends HttpServlet{

}
Test is non pojo class

Eg6: class Test implements java.sql.connection{

}
"Test" is non pojo class

Eg7: class Test{
public void m1(Statement st){

}
}
Test is pojo class

Eg8 : class Test extends Demo {

}
class Demo implements java.rmi.Remote{

}
Demo, Test are not pojo classes

Eg9 :
@Entity (annotation of HIbernate)
class Test{

}
Test is a pojo class

Note Do not consider annotations and api used inside the methods of a class to check whether class is POJO or Not.

5/6/15
POJI (plain old java interface)

→ It indicate an interface with out specialities, ie, an ordinary interface.
→ The interface that is not extending from Technology/ Framework api interfaces is called "POJI."

Eg.1) interface Test{

}
Test is POJI

2) interface Test extends Serializable {

}
Test is POJI

3) interface Test extends java.rmi.Remote{

}
Test is not POJI

4) interface Test extends Demo{

}
interface Demo{

}
Demo, Test is POJO.

→ The framework that supports POJO, POJI modal programming are called "Non-Invasive or light weight framework".

Eg: Hibernate, spring, struts etc..

Java Bean

→ It is a java class that is developed with some standards.

a) Must be public class and should not be final or abstract class.
b) Can implement serializable (I) if needed
c) Must have direct or indirect zero (0)-param construction
d) Member variable are called "bean properties" and they must be taken as private.
e) Every bean properties should have getter (read) & setter (write) methods (public)

Eg:
```
public class StudentBean{
    //bean properties
    private int sno;
    private String sname;
    //setters and getters
    ---
    ---
}
```

→ In realtime, JavaBean is used as helper class to represent data with multiple values in the form of obj and to send that obj from one layer to another layer.

⇒ Java Bean object is acceptable format to send and receive data across the multiple layers irrespective of the technologies (or) frameworks that are used in the layers to develop logics.

⇒ The Java Bean class whose object holds input values given by client (or) end user is called "VO" class (Value Object class)

⇒ The Java Bean class whose object needed for persistence operations (DAO class) is called "BO" class (Business object class)

⇒ The Java Bean class whose object can be sent over the network is called "DTO" class

BO class : Business Object class
VO class : Value object class
DTO class : Data transfer Object class

Bean / Component classes

⇒ It is a java class that contains member variables and business methods having business logic manipulating the data of member variables.

```
class CardProcessor {
    private int cardNO;
public int processCard(int cardNO){
        ---  //business logic
    }
public boolean blockCard(int cardNO){
        ---  // business logic
    }
}
```

The Bean / component class can extend and implement any class or interfaces.

Spring Bean

⇒ The Java class whose object can be managed by spring container is called "Spring Bean".

⇒ Spring Bean can be pojo class, Java bean class, Bean / component class but its object must be created and managed by spring container

⇒ Spring Bean can be user-defined class / pre-defined class / third party class.

⇒ We can not take abstract class / interface as Spring Bean.

⇒ To make java class as Spring Bean it must be configured in spring Bean configure file (xml) to make spring container to create and managed that spring Bean class object.

Important points

1) Spring Bean can be pojo class ? (yes)
2) Every Java Bean is POJO class ? (yes)
3) Every POJO class is Java Bean ? (no)
4) Every spring Bean is Java Bean ? (no)
5) Spring Bean can be a Java Bean ? (yes)
6) Spring Bean can not implement spring api interfaces ? (no)
7) Any component / Bean class can be taken as spring Bean ? (yes)

→ Spring gives two light weight containers which can be started in any application by just creating object to container classes.

⇒ Any java class / Appln that can manage the life cycle of given resource is called "Container"

⇒ Spring containers are

a) Bean factory (Basic container)

(performs Spring Bean management and dependency injection)

b) Application context (Advanced container)

(Internally uses BeanFactory container) It can perform advanced operations along with Bean management and dependency injection).

→ To activate/start BeanFactory container we should create obj of a class that implements "BeanFactory(I)";

Eg BeanFactory factory = new XMLBeanFactory (Resource obj)

↓ Implementation class of BeanFactory Interface.

↳ points to spring bean cfg file (xml file)

→ Spring containers are not replacement/alternate for servlet, JSP containers (heavyweight).

06/06/15

Spring Core Module

⇒ Base module for other spring modules

⇒ Gives two IOC containers/spring containers

a) Bean Factory

b) Application context

⇒ When it is used alone we can develop standalone Apps

⇒ When it is used along with other technologies we can develop model layer, business layer by taking the advantage of Dependency Injection

⇒ It simply talks about Spring Bean lifecycle management and Dependency Injection.

Dependency Injection

⇒ If the underlying server/container dynamically assigns the dependent values to our resources(class/object) then it is called "Dependency Injection".

⇒ There are multiple ways of performing Dependency Injection.

a) setter Injection (Container calls setXxx(-) to inject dependent values)

b) Constructor Injection (Container uses parameterized constructor to create bean class obj and to inject values to that obj)

c) InterfaceInjection (Aware Injection)

d) Method Injection

e) LookupMethod Injection

and etc....

1) Setter Injection sample code

Spring Bean class

WishGenerator.java

Spring Bean class

```
package com.nt.bean;

public class WishGenerator {
    //bean properties
    private String name;
    private Date date;

    //setXxx(-) methods for setter injection
    public void setName(String name){
        this.name =name;
    }
    public void setDate(Date date){
        this.date =date;
    }

    public String generateWishMsg(){
        //get hour of Date and time
        int h=date.getHour();
        if(h<12)
            S.O.P("Good Morning"+name);
        else
            S.O.P("Good Evening"+name);
    }
}
```

applicationContext.xml (Spring Bean Cfg file | Spring cfg file)

```
..... DTD/XSD statements
.....
<beans>
    <bean id="dt" class="java.util.Date"/>
    <bean id="wsg" class="com.nt.bean.WishGenerator">
        <property name="name" value="ramesh"/>
        <property name="date" ref="dt"/>
    </bean>
</beans>
```

Spring cfg file

⇒ Any <filename>.xml can be taken as spring bean config file, but it is recommended to take applicationContext.xml in standalone spring apps.

⇒ In SpringBean cfg file we configure,

a) classes as spring beans
b) dependency injection cfgs
c) Spring bean life cycle cfgs
and etc...

⇒ We must supply spring Bean cfg file while activating spring container.

⇒ Every Spring is identified with its bean <id>.
<property> used to config bean property for setter injection.

⇒ In the above configs, Spring container creates "Date", "WishGenerator" class objs using 0-param constructors and setName() and setDate() methods to perform setter injection.

2) Constructor Injection Sample Code

WishGenerator.java

```
package com.nt.beans;
    public class WishGenerator{
        //bean properties
      private string name;
      private Date date;
      // param constructor for constructor injection.
      public WishGenerator(string name,Date date){
          this.name = name;
          this.date = date;
       }
      ... // Business method
     }
```

applicationContext.xml

```
<!-- DTD & XSD
-->
<beans> <bean id="dt" class="java.util.Date" />  ← it will generate 0-param constructor by default to Date.
    <bean id="wsg" class="com.nt.beans.WishGenerator" >
    <constructor-arg value="raja" />
    <constructor-arg ref="dt" />
    </bean>
</beans>
```

comment: Based on above configuration, the spring container uses 0-param constructor to create Date class object and two-param constructor to create WishGenerator class object because <constructor-arg> tag is placed 2 times in the xml file.

⇒ In this process injection will also be completed.

08/08/15

Procedure to develop spring core module Appln that performs setter injection.

STEP 1) Keep Eclipse IDE ready ⇔ Luna

Eclipse
↦ type : IDE for Java
↦ Version : 4.x (Luna)
↦ Vendor : Eclipse
↦ Open source
↦ To download SW : www.eclipse.org
(as a zip file : Eclipse-jee-Luna-SR1-win32-zip)
↦ To install Eclipse : Extract zip.

STEP 2) Design the application.

IOC proj 1 (Setter Injection)
↦ src
↦ com.nt.cfgs
↦ applicationContext.xml (Spring bean cfg file)
↦ com.nt.beans
↦ WishGenerator.java
↦ com.nt.test
↦ ClientApp.java

STEP 3) Develop the resource.

(Workspace : D:/NTSP27)

File → New → project → Java project → New → IOC Proj 1.

//WishGenerator.java POJO class in com.nt.beans.

```
package com.nt.beans;

public class WishGenerator {

    //bean properties
    private String name;
    private Date date;        (ctrl+shift+o → gives package import stmts)
```

```
//setter methods for setter injection
public void setName(String name){
    this.name=name;
}
public void setDate(Date date){
    this.date =date;
}
//Business method
public String generateWishMsg(){
    return "Good morning:"+name+"-->"+date;
}
}
```

2) applicationContext.xml file in com.nt.cfgs package

```
<beans xmlns="http://www.framework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchemainstance"
  xsi:schemaLocation="http://www.springframework.org/schemas/begins
    http://www.springframework.org/schema/beans/spring-beans.xsd">
    <bean id="dt" class="java.util.Date"/>
    <bean id="wsg" class="com.nt.beans.WishGenerator">
        <property name="name" value="raja"/>
        <property name="date" ref="dt"/>
    </bean>
</beans>
```

Note: XSD(or) DTD are the two techniques that can be used to define building blocks(or) rules for construction of XML document.

3) ClientApp.java

```
package com.nt.test;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.FileSystemResource;
import com.nt.beans.WishGenerator;

public class ClientApp{
  public static void main(String[] args){
    //Locate spring bean cfg file
FileSystemResource resource = new FileSystemResource("src/com/nt/cfgs/applicationContext.xml");
    //Activate BeanFactory container
XmlBeanFactory factory = new XmlBeanFactory(resource);
    //Get Bean class object
WishGenerator bean = (WishGenerator) factory.getBean("wsg");
    //call Business method
String result = bean.generateWishMsg();
      S.o.p("result");
  }//main method
}//class
```

4) Add the following jar files to build path of the project.

Right click on project → Build path → Configure Build path → Libraries → Add External jars →

commons-logging.jar → (collect from spring 2.5)

spring-beans-version.jar
spring-context-version.jar
spring-context-support-version.jar
spring-core-version.jar
spring-expression-version.jar
→ (collect from spring 3.x/4.x libs folder)

5) Run the client Application

o/p: Good morning raja --> Tue 09 June 2015.

09/08/15

Comments

1) FileSystemResource res = new FileSystemResource ("src/com/nt/cfgs/applicationContext.xml");

→ Gives Resource object using java.io.File(class) internally pointing to the path and filename of spring bean cfg file.

2) XmlBeanFactory factory = new XmlBeanFactory (resource);

(or)

BeanFactory factory = new XmlBeanFactory (resource);

a) Based on the given "resource" object it locates the spring beans cfg file (xml) from the specified path of the file system.

b) checks whether that xml file is well-formed and valid document or not.

Note: If xml file satisfies the syntax rules then it is called "well-formed Document".

Note: If xml file satisfies XSD(or)DTD rules then it is called "valid Document".

c) creates an empty logical IOC container in JVM memory.

d) Uses the sax parser (xml parser) to xml file (bean cfg file) data and maintains it as InMemory MetaData (or) InMemory Cfg.

e) Returns XmlBeanFactory class object ref representing IOC container.

3) WishGenerator bean = (WishGenerator) factory.getBean("wsg");

a) Loads "com.nt.bean.WishGenerator" class from InMemory MetaData based on the given "wsg" and creates obj for it.

Class.forName("com.nt.beans.WishGenerator").newInstance();

b) observes "name" with <property> tag and calls setName("raja") method on WishGenerator class obj.

c) Observes "date" with <property> tag and loads "java.util.Date" class to create the obj.

Class.forName("java.util.Date").newInstance();

Also setDate(Date Class obj) on WishGenerator class obj to perform setter injection.

d) Keeps Date class obj and WishGenerator class obj in IOC container as elements in HashMap.

HashMap

key	value
dt	Date class obj (dt)
wsg	WishGenerator class obj (wsg)

(beanId) (Bean class obj)

e) Return WishGenerator class obj reference back to Client Appln.

comments

1) If we configure both setter injection & constructor injection on bean property, the values given by setter injection will remain the final values of injection because setter method executes after constructor execution and overrides the values injected by the constructor injection.

2) If spring bean class is configured with out any injection (or) only with setter injection on (one or more properties) then container uses zero-param constructor to create bean class obj. If any property is configured for constructor injection then it uses parameterized constructor to create bean class object.

3) In spring 3.x, bean properties and its setter methods can be taken as static. But in spring 2.x, we can not take setter methods as static.

4) WishGenerator bean = (WishGenerator) factory.getBean("wsg"); [Type casting is mandatory]

(or)

WishGenerator bean = factory.getBean("wsg", WishGenerator.class);

(We can avoid type casting).

10/6/15

Prototypes

```
public Object getBean(String id);
public <T> getBean(String id, <T>class);
```

5) What is difference b/w FileSystemResource and ClassPathResource?

⇒ FileSystemResource can locate given spring bean cfg file from the specified path of the fileSystem. Here we can pass either absolute path or relative path.

```
FileSystemResource res =
    new FileSystemResource("src/com/nt/cfgs/applicationContext.xml");
```

⇒ ClassPathResource can locate given spring bean cfg file from directories (or) jar files that are added to classpath | Build path.

ClassPathResource res = new ClassPathResource("applicationContext.xml"); (But add com.nt.cfgs pkg to build path)

Rightclick on project → buildpath → cfg build path → libraries → Add class folder → ___

Understanding "collaboration" & tight coupling, loose coupling:

⇒ In real time projects development, we can not develop entire logics in one class. We need to take multiple classes and these classes will maintain dependency with each other.

⇒ The class that uses other class logics / methods is called "Target class" and the class that acts as helper class is called "dependent class" and we can say both classes are in collaboration.

→ Flipkart and DTDC classes must be there in collaboration ie., flipkart is Target class / Main class and DTDC class is dependent class.

→ To make Target class / Main class using the logics of Dependent class we can use following approaches.

Approach 1 : Create Dependent class obj in Target class.

Approach 2 : Create Dependent class obj in Factory class and make Target class using that factory class.

Approach 3 : Target class gets Dependent class obj from jndi registry.

Approach 4 : Target class extends from Dependent class

Approach 5 : Make underlying container injecting Dependent class obj to Target class.

Approach 1

→ Here Target class creates the object of Dependent class uses that object. This is nothing but composition (classes containing "HAS A" relationship).

Target class

```
class Flipkart{

    DTDC dtdc = new DTDC();

    purchase(){
      dtdc.deliver();
    }
}
```

Dependent class

```
class DTDC{
    deliver(){
    ----
    }
}
```

Limitations

a) Instead of DTDC courier, if we need to use BlueDart or FirstFlight courier we need to modify Target class (Flipkart)

b) If ; deliver() method is changed to supply() method in DTDC class. we need to modify the Flipkart class.

→ The above limitations indicates Target and Dependent classes are tightly coupled.

Approach 2

⇒ The class that can create and return 1 or more objects is called "Factory class".

Factory class.

```
public class DTDCFactory{

  public static DTDC getInstance(){

    return new DTDC();
  }
}
```

DTDC object is created in factory class

Dependent class

```
class DTDC{

  deliver(){
    ---
  }
}
```

Target class.

```
class Flipkart{

    purchase(){
    DTDC dtdc = DTDCFactory.getInstance();
        dtdc.deliver();
    }
}
```

Limitations

a) If Flipkart wants to use Bluedart then it needs BlueDart Factory class instead of DTDCFactory class.

b) If deliver() method is changed to supply() method in DTDC class. we need to modify the Flipkart class.

Note : Here also tight coupling is observed.

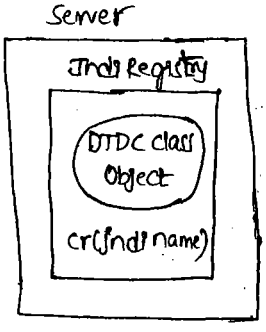
Approach 3

→ If u want to provide global visibility to object or object reference then it should be placed in jndi registry.

Eg: rmi registry, cos registry, weblogic registry etc..

Note : Every Webserver / Application server gives one built in JNDI registry.

```
class Flipkart{
    Purchase(){
      InitialContext ic = new InitialContext();
      DTDC dtdc = (DTDC)ic.lookup("cr");
        dtdc.deliver();
    }
}
```



Advantages

1) Because of jndi registry DTDC object gets global visibility
2) Flipkart class can use the already available DTDC class object.

Limitations

a) If Flipkart wants BlueDart object instead of DTDC object we need to place that obj in jndi registry.
b) If deliver() method is changed to supply() method in DTDC class. we need to modify the Flipkart class.

Note It also gives tight coupling.

11/06/15

Approach4

⇒ Make Target class extending from dependent class

```
class DTDC{
    deliver(){
    ---
    }
}
```

```
class Flipkart extends DTDC{
    purchase(){
      deliver();
    }
}
```

Limitations

a) Flipkart class cannot extends more than one dependent classes.
b) In order to change dependency from DTDC to BlueDart we need to make Flipkart class extending from BlueDart class
c) If deliver() method is changed to supply() method in DTDC class. we need to modify the Flipkart class.

Note shows Tight coupling

Approach5

⇒ Use dependency Injection (setter injection | constructor injection)

```
class Flipkart{
    private DTDC dtdc;
  public void setDtdc(DTDC dtdc){
     this.dtdc = dtdc;
  }
  purchase(){
  ---
    dtdc.deliver();
  }
}
```

```
class DTDC{
    deliver(){
    ---
    }
}
```

→ This Approach5 is much similar to Approach 1 (or) Approach2. The only difference is the underlaying container is managing dependency in Approach5, where as programmer is managing dependency in Approach 1, 2.

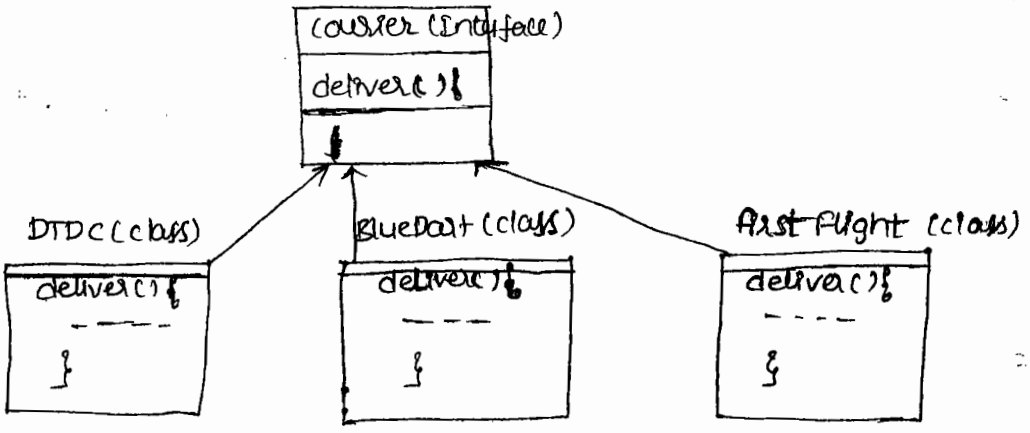
Limitations : same as approach 1

(a) Instead of DTDC courier, if we need to use BlueDart(or) FirstFlight courier we need to modify Target class (Flipkart).

(b) If deliver() method is changed to supply() method in DTDC class. we need to modify Target class (Flipkart).

Note: shows tight coupling.

→ Approach5 is good towards managing dependency but inorder to achieve loose coupling we should not directly work with dependent classes in target class. It is recommended to make all dependent classes implementing common interface having common method declaration so that we can start using this common interface in target class.




```
public class Flipkart {
    private Courier courier;
    public void setCourier(Courier courier){
        this.courier = courier;
    }
    purchase() {
        courier.deliver();
    }
}
```

→ The above Flipkart class is not working with implementation class names, It is working with interface. So we can make underlying container injecting any implementation class object to Flipkart (Target class).

→ All Dependent classes are implementing common interface courier so there is no possibility of changing method name deliver().

NOTE: The above code shows loosely coupling between target class (Flipkart) and dependent classes.

⇒ To achieve loosely coupling in spring use,

a) Dependency Injection
b) Interface Model programming (POJI Model)

IOC Proj2 (loose coupling)

- src
  - com.nt.beans
    - Flipkart.java, courier.java, BlueDart.java, DTDC.java
  - com.nt.cfgs
    - applicationContext.xml
  - com.nt.test
    - LCTestApp.java

Jars : same

Courier.java (Interface)

```
package com.nt.beans;

public interface courier{

    public string deliver(int orderId);

}
```

BlueDart.java DTDC.java

```
package com.nt.beans;

public class DTDC implements Courier{

    public string deliver(int orderId){

        return "DTDC is ready to deliver product is "+orderId;

    }

}
```

DTDC.java BlueDart.java

```
package com.nt.beans

    public class BlueDart implements Courier{
        public String deliver(int orderId){
            return "BlueDart ready to deliver product is "+ orderId;
        }
    }
```

Flipkart.java

```
package com.nt.beans;
  public class Flipkart{
      private Courier courier;
    // setter method for setter injection
    public void setCourier(Courier courier){
        this.courier = courier;
    }
     public String purchase(String item[]){
        Random rad = new Random();
          int orderId = rad.nextInt(7000);
    String status = courier.deliver(orderId);
        return "Bill amount for "+ orderId +" is 7000 "+ status;
    }
  }
```

applicationContext.xml

```
<beans>----->
<bean id="dtdc" class="com.nt.beans.DTDC"/>
<bean id="bluedart" class="com.nt.beans.BlueDart"/>
<bean id="fk" class="com.nt.beans.flipkart">
    <property name="courier" ref="dtdc"/>
  </bean>
</beans>
```

LCTestApp.java

```
package com.nt.test;
  public class LCTestApp{
public static void main(String[] args){
//Create BeanFactory Object to create IOC container
BeanFactory factory = new XmlBeanFactory(new FileSystemResource("src\com\nt\cfgs\app.xml"));
// Get Bean object
Flipkart bean = factory.getBean("fk", Flipkart.class);
String billmsg = bean.purchase(new String[]{'shirt', 'mobile'});
System.out.println(billmsg);
}}
```

15/06/15

In spring bean configuration file

1) <property name="msg" value="hello" />

can be written as

```
<property name="msg">
  <value> hello </value>
</property>
```

2) <property name="date" ref="dt" />

can be written as

```
<property name="date">
  <ref bean="dt" />
</property>
```

Attaching source code and API docs with eclipse IDE

⇒ Right click on project → Build Path → configure build path.

Select any jar file that contains the classes that we are using.

1) source attachment → Edit → External location → path → External file

select spring/libs/spring-beans-4.1.6.RELEASE-sources.

2) Javadoc location → Edit → Javadoc in archeive → External file → Archeive Path

(select spring software ZIP file) → spring-framework-4.1.6.RELEASE/docs/javadoc-api →

Validate → ok.

Creating user-defined library (spring IOC) in eclipse IDE

⇒ Right click on project → Build Path → configure build path → Add Library →

User Library → user libraries → New → IOC Lib → Add External Jars →

Required jars from framework folder.

Design Patterns

⇒ Design patterns are set of rules which come as best solutions for reoccuring problems of Application development.

⇒ Design patterns are best practises for utilizing sw languages, technologies and frameworks in application development.

eg Singleton, Factory and etc..

⇒ Bean Factory IOC container is designed based on Factory pattern...

⇒ It is always recomanded to develop spring applns based on strategy pattern.

Strategy Pattern

⇒ While developing multiple classes having dependency it is recomanded to follow this design pattern.

⇒ This pattern is not spring pattern it can be used any where since the spring manages the dependency between classes, it is recomanded to use this pattern while developing spring applns.

⇒ strategy implementation is all about implementing 3 principles.

a) Prefer Composition over Inheritance

b) Always code to interfaces and never code with implementation classes.

c) code should be open for extension and must be closed for modifications

a) Prefer composition over Inheritance

(1) If one class creates the object of another class to use its logics then it is called "composition" (Has-a relationship)

```
class A        class B {
{ ---            A a = new A();
}                }
```

If one class extends from another class then it is called "Inheritance". (is-a relationship)

```
class A        class B extends A
{ ---          {
}                B b = new B();
               }
```

⇒ Inheritance is having following limitations in Java.

a) It doesnot support multiple Inheritance

b) code become easily breakable

eg:- 1) class c extends A, B (wrong)

```
{ ---
}
```

2) class c (correct) ✓

```
{
  A a = new A();
  B b = new B();
}
```

code is easily breakable

```
class Test{
  public int x(){
    ----
    return 100;
  }
}
class Demo extends Test{
  public int x(){
    ---
    return 1000;
  }
}
```

⇒ If we change return type (or) parameter types of x() in Test, we cannot compile Demo class because overriding rules. (Let use "Demo", "Test" classes given by two different developers)

solution

```
class A{
  public int x(){
    ---;
    return 100;
  }
}
```

```
class B{
  A a = new A();
  public int y(){
    int result = a.x();
  }
}
```

⇒ If x() method return type is changed in "A" there is no need of doing major modifications in "B"!

⇒ If x() return type is changed to float then we need to write y() in B as shown below.

```
public int y(){
  float res = a.x();
  ----;
}
```

(11) Always code to interfaces and never code to implementations

Problem

```
class A{
  ---
 }
class B{
 ---
 }
class Test{
A a=new A();
--
}
```

Eg. (1) Assigning class "B" object instead of class "A" object in "Test" class is not possible with minimum modifications.

solution

```
Interface X(){
  ---
 }
class A implements X{
  ---;
 }
class B implements X{
  --;
 }
class Test{
 X x=new A();
  (or)
 X x=new B();
}
```

⇒ When we code with interfaces we can achieve loose coupling between target class dependent class;

16/06/15

Eg (2)

16/06/15

Eg(2) With out modifying the code "Test" class we assign either the object of class A (or) class B, this gives loose coupling.

```
Interface x{
   ---;
  }
class A implement x {        class B implements x{
   ---;                          ---;
  }                             }
class Test {
  x x;
 public void setX(x x){
     this.x=x;
   }
}
Test t=new Test();
  t.setx(new A());
  (or)
  t.setx(new B());
```

(iii) Code must be open for Extension and must be closed for modification

⇒ If we follow 2nd principle perfectly by taking the methods of implementation classes A,B as final methods our code becomes closed for modifications. Similarly it allows to write more implementation classes for interface 'x' having new logics, this makes our code open for extension.

Observing strategy pattern in Application NO:2

a) Flipkart class wants the logics of DTDC (or) BlueDart classes but it is not inheriting from those classes but getting objects of those classes through composition (principle(1)).

b) BlueDart, DTDC classes are implementing "courier(Interface)" & their objects are assigned to Flipkart class through this "courier(Interface)" (principle(2)).

c) By making the deliver() method of BlueDart, DTDC classes as final method we can make that code closed for modification and we can develop new implementation class for "courier(Interface)" if new courier service (like first flight) is required for "Flipkart" class. (principle(3)).

Resolving / Identifying parameters in constructor injection

1) We generally configure the dependent values of constructor injection in the order the parameters of constructor are available. But we can mismatch this order during the configuration, we can identify the parameters by using index or type or name.

Ex(1) Resolving based on parameter type

```
class student{
 private int sno;
 private String sname;
 private float avg;

public student(int sno,String sname, float avg){
   this.sno=sno;
   this.sname=sname;
   this.avg=avg;
 }
  ll toString(){
      --
     }
}
```

application context.xml

```
<bean id="st" class="student">
 <constructor-arg value="raja" type="java.lang.String"/>
 <constructor-arg value="1000" type="int"/>
 <constructor-arg value="60.98" type="float"/>
<\bean>
```

Limitation If multiple parameters are having same type then this approach cannot be used.

Ex(2) Resolving based on Index

```
public class Marks{
    private int m1;
    private int m2;
    private int m3;
 ll 3 param constructor
```

```
public Marks (int m1, int m2, int m3) {
    this.m1 = m1;
    this.m2 = m2;
    this.m3 = m3;
  }
  //toString()
 }
```

applicationContext.xml

```
<bean id="mk" class="Marks">
  <constructor-arg value="30" index="2" />
  <constructor-arg value="20" index="1" />
  <constructor-arg value="10" index="0" />
</bean>
```

Eg(3)

```
public class Machine {
    private int id;
    private string name;
    private Date dt;
    public Machine (int id, string name, Date dt){
      this.id = id;
      this.name = name;
      this.dt = dt;
    }
    //toString(){
      --
    }
}
```

applicationContext.xml

```
<bean id="dt" class="java.util.Date" />
<bean id="m1" class="com.nt.beans.Machine">
  <constructor-arg ref="dt" />
  <constructor-arg value="ramesh" />
  <constructor-arg value="101" index="0" />
</bean>
```

Ex (4) Resolving constructor parameters through Names

```
public class faculty {
  private int id;
  private string name, sub;
@)Constructor Properties ( { "no", "fname", "subject" } ) // Not required from 4.x.
 public faculty ( int no, string fname, string subject) {
        id=no ;
       name =fname;
       sub = subject;
     }
  //toString(){
     --
     }
}
```

application context.xml

```
<bean id="f1"  class ="com.nt.beans.Faculty" />
   <constructor-arg   name="subject"   value ="java" />
   <constructor-arg   name="fname"     value="raja" />
   <constructor-arg   name="no"        value = "678" />
 </bean>
```

⇒ Note:

⇒ In spring 3.x, the IOC container cannot get parameter names of constructor directly until the parameter names are configured by using @)constructor properties.

⇒ But from spring 4.x, the IOC container can recognise parameter names of the constructor directly.

```
package com.nt.beans;
import java.beans.ConstructorProperties;
```

Faculty

```
public class Faculty {
    private int id;
    private String name, subject;
    @ConstructorProperties({"no", "fname", "sub"})
    public Faculty(int no, String fname, String sub){
        this.id=no;
        this.name=fname;
        this.subject=sub;
    }
    public String toString(){
        return "Faculty [id="+id+", name="+name+", subject="+subject+"]";
    }
}
```

Machine

```
public class Machine {
    private int id;
    private String name;
    private Date dom;
    public Machine(int id, String name, Date dom){
        this.id=id;
        this.name=name;
        this.dom=dom;
    }
    @Override
    public String toString(){
        return "Machine [id="+id+", name="+fname+", dom="+dom+"]";
    }
}
```

Marks

```
public class Marks {
    private int m1, m2, m3 ;
    public Marks (int m1, m2, int m3) {
     this.m1 = m1;
     this.m2 = m2;
    this.m3 = m3;
    }
    @override
    public string toString() {
    return "Marks [m1 = "+m1 + ", m2= "+m2+" , m3 ="+m3+"] ";
     }
}
```

Student

```
public class student {
    private int sno;
    private float avg;
    private string sname;
    public student ( int sno, string sname, float avg) {
    s.o.p(" student 3-param constructor ");
        this.sno = sno;
        this.sname = sname;
        this.avg  =avg ;
        }
    @ overrides
    public string toString() {
    return "student [sno="+sno+ ", sname ="+sname+", avg="+avg + "] ";
    }
}
```

applicationContext.xml

```
<bean id="st" class="com.nt.beans.Student">
 <constructor-arg value="1001" type="int" />
 <constructor-arg value="89.45f" type="float" />
 <constructor-arg value="raja" type="java.lang.String" />
</bean>

<bean id="mk" class="com.nt.beans.Marks">
 <constructor-arg value="40" index="2" />
 <constructor-arg value="20" index="1" />
 <constructor-arg value="10" index="0" />
</bean>

<bean id="dt" class="java.util.Date" />
<bean id="m1" class="com.nt.beans.Machine">
    <constructor-arg ref="dt" />
    <constructor-arg value="ramesh" />
    <constructor-arg value="101" index="0" />
</bean>

<bean id="f1" class="com.nt.beans.faculty">
 <constructor-arg name="sub" value="java" />
 <constructor-arg name="fname" value="rajesh" />
 <constructor-arg name="no" value="678" />
</bean>
```

ClientApp.java

```
public class ClientApp{
   public static void main(String[] args){
     //create BeanFactory obj
  BeanFactory factory = new XmlBeanFactory(new FileSystemResource("src/com/nt/cfgs/
                                          applicationContext.xml"));
        Student st = (Student) factory.getBean("st");
           S.O.P(st);
        Marks mk = (Marks) factory.getBean("mk");
           SO.P(mk);
       Machine m1 = (Machine) factory.getBean("m1");
           S.O.P(m1);
     Faculty f1 = (Faculty) factory.getBean("f1"); SO.P(f1); } }
```

8/6/15

Cycle Dependency Injection

⇒ If two beans are dependent to each other then we can say they are in cycle dependency. Spring container cannot perform cycle dependency injection through constructor injection can perform through setter injection.

constructor-injection

Problem:

```
class A {
    B b;
    public A(B b) {
        this.b = b;
    }
}
class B {
    A a;
    public B(A a) {
        this.a = a;
    }
}
```

In applicationContext.xml

```
<bean id="a" class="A">
    <constructor-arg ref="b" />
</bean>
<bean id="b" class="B">
    <constructor-arg ref="a" />
</bean>
```

Solution: (using setter Injection)

```
class A {
    B b;
    public void setB(B b) {
        this.b = b;
    }
}
class B {
    A a;
    public void setA(A a) {
        this.a = a;
    }
}
```

In applicationContext.xml

```
<bean id="a" class="A">
    <property name="b" ref="b" />
</bean>
<bean id="b" class="B">
    <property name="a" ref="a" />
</bean>
```

ClientApp.java

```
BeanFactory factory = new XmlBeanFactory(new FileSystemResource
        ("applicationContext.xml"));
A a = factory.getBean("a", A.class);
S.O.P(a.hashCode());
```

When should we go for setter Injection and when should we go for Constructor Injection

1) If bean class contains only one property or If all the properties of bean class should participate in dependency Injection then go for constructor Injection because this injection is faster than setter injection.

2) If bean class contains more than 1 property and there is no need of making all properties participating dependency injection then we go for setter injection.

Eg: If bean class contains 4 properties and there is no need of making all properties participating in dependency injection then we need to write max of 4 setter methods to perform setter injection, But we need to write 4!(24) constructors to support constructor injection in all angles..

Difference bt setter Injection & Constructor Injection

Setter Injection	Constructor Injection
→ Performs injection after creating Bean class object so bit slow in injection.	→ Performs injection while creating Bean class object so no delay in injection
→ Supports cyclic dependency injection	→ Does not support cyclic dependency injection
→ <property> is required	→ <constructor-arg> is required
→ To perform setter injection on 'n' properties 'n' setter methods are required	→ To perform constructor injection on 'n' properties in all angles 'n!' constructors are required.
→ If all properties are configured for setter injection then container creates bean class objects using zero param constructor	→ If any property is configured for constructor injection container creates Bean class object using parameterised constructor.

Inner Bean

→ This is no way related with inner classes of Java. It is all about cfg one bean as the inner bean of another bean by placing <bean> tag inside the <property> or <constructor-arg> tag.

```
<bean id="fpk" class="com.nt.beans.Flipkart" >
  <property name="courier">
  <bean id="dtdc" class="com.nt.beans.DTDC"/> <!--inner bean-->
  </property>
</bean>
```

→ Use inner bean cfg if want to make bean as dependent bean to another bean only for 1 time.

Limitations with inner beans

a) Inner bean cannot be used as dependent bean of multiple other beans

b) Inner bean object can not be accessed from client apps.

Note : bean id is optional for inner beans.

Can we configure spring bean class with out bean id?

Ans Yes, possible because container generates default bean id in the following notation.

<pkg> . <Beanclass>#n (where 'n' is 0,1,2,--)

eg.

<bean class="com.nt.beans.Flipkart"> bean id is "com.nt.beans.Flipkart#0"

</bean>

<bean class="com.nt.beans.Flipkart"> bean id is "com.nt.beans.Flipkart#1"

</bean>

Spring supports dependency injection on the following properties

a) simple properties (primitive data type, string)

b) Reference / object type properties

c) collection properties. (ArrayList, Set, Map, Properties and etc..)

Property type	tag for dependency injection cfg
1) simple property	value attribute on <value> tag
2) Reference property	ref attribute (or) <ref> tag
3) ArrayList	<list>
4) java.util.List	<list>

Property tag	tag for dependency injection configuration
5) java.util.Set	<set>
6) java.util.Map	<map>
7) java.util.Properties	<props>

Injecting Array

```
public class student {
    private String colors[];
public void setColors(String colors[]){
    this.colors = colors;
  }
 //to string()
 ---
}
```

In applicationContext.xml

```
<bean id="st" class="com.nt.student">
  <property name="colors">
    <list>
      <value> Red </value>
      <value> Blue </value>
      <value> Green </value>
    </list>
  </property>
</bean>
```

Internal code

```
student st = (student) Class.forName("student").newInstance(); ...
    String colors[] = new String [] {"red", "green", "blue"};
      st.setColors(colors);
```

19/6/15

Comment For example application on injecting values to different types of collection properties refer application 6&7 of page No: 45 to 49.

Injecting List

```
public class student{
    private List<String> subjects;
        public void setSubjects(List<String> subjects){
            this.subjects = subjects;
        }
        //toString()
        System.out.println(names.getClass());
    }
```

Weapon to check the class Names

```
private List<String> names;
public void setNames(List<String names){
    this.names = names;
}
```

applicationContext.xml

```
<bean id="st" class="com.nt.beans.student">
  <property name="names">
    <list>
      <value>raja</value>
      <value>ravi</value>
      <value>raja</value>
    </list>
  </property>
</bean>
```

Note: java.util.List allows duplicates so we can inject duplicate values.

Internal Code

```
student st = (student) Class.forName("com.nt.beans.student").newInstance();
List<String> names = new ArrayList<String>();
    names.add("raja");
    names.add("ravi");
    names.add("raja");
    st.setNames(names);
```

Injecting set

Note set cannot allow duplicate values.

eg.

```
public class User {
        private set <String> phones;

        public void setPhones (Set<String> phones) {
            this.phones = phones;
        }
        //toString()
    }
```

application context.xml

```
<bean id="user" class="com.nt beans.User">
  <property names="phones">
     <set>
        <value> +91-9855997788 </value>
        <value> +91-8877990099 </value>
        </set>
      </property>
  </bean>
```

Note It internally uses hash code to complete the injection.

Injecting Map.

→ Each element of map allows one key, one value. use <map> tag to configure injection.

eg.

```
public class college {
    private Map<String, String> facultySubjects;

    public void setfacultySubjects ( Map<String, String> facultySubjects) {
        this.facultySubjects = facultySubjects;
    }
    //toString()
}
```

applicationContext.xml

```
<bean id="clg" class="com.nt.beans.College">
  <property name="facultySubjects">
    <map>
      <entry key="raja" value="java" /> ||element1
      <entry> || element2
        <key><value>ravi</value></key>
        <value>.net</value>
      </entry>
      <entry key="ramesh"> || element3
        <value>SAP</value>
      </entry>
    </map>
  </property>
```

comment

⇒ Map collection elements allows any objects as keys and any objects as values. But keys cannot be duplicates but values can be duplicated.

eg:
```
public class college{
    private Map<?,?>facultyPhones;
public void setFacultyPhones(Map<?,?>facultyPhones){
  this.facultyPhones = facultyPhones;
  }
  //toString();
}
```

internal code

```
mapobj.push("key","value");
```

applicationContext.xml

```
<bean id="dt" class="java.util.Date">
<bean id="clg" class="com.nt.beans.college">
  <property name="facultyPhones">
    <map>
      <entry key="raja" value-ref="user" />
      <entry key-ref="dt" value-ref="user" />   (or)
    </map>                    <!--<entry><key> ref bean="dt" /></key>
  </property>                     </entry>--> ref bean="user" />
</bean>
```

Injecting Properties

⇒ java.util.Properties allows only strings as the key-value pairs, ie, key must be string and values must be string.

Eg.

```
public class EmployeeProfile {
    private Properties empDesgs;
    public void setEmpDesgs(Properties empDesgs) {
        this.empDesgs = empDesgs;
    }
    // toString();
}
```

applicationContext.xml

```
<bean id="emp" class="com.nt.beans.EmployeeProfile">
    <property name="empDesgs">
        <props>
            <prop key="anil">clerk</prop> // element1
            <prop key="johny">SE</prop> // element2
        </props>
    </property>
</bean>
```

2016/15

Injecting Null (<null/>)

⇒ Given to inject null value to reference type bean property.

⇒ useful in constructor injection to inject null value to certain ref type parameter when we cannot supply value.

⇒ In constructor injection we need to supply values to all the parameters compulsory.

⇒ If we cannot supply values to certain params then we can use <null/> tag to inject null value...

Eg: public class user{

```
    private int id;
    private String name;
    private Date dob;
    //3-param constructor
```

```
public User(int id, String name, Date dob){
    this.id = id;
    this.name = name;
    this.dob = dob;
}
//toString();
--
}
```

applicationContext.xml

```
<bean id="user" class="com.nt.beans.User">
    <constructor-arg value="1001" />
    <constructor-arg value="raja" />
<constructor-arg><null/> <!-- Null Injection --> </constructor-arg>
</bean>
```

clientApp.java

```
Public class Test{
    public static void main(String[] args){
        BeanFactory factory = new XmlBeanFactory(new FileSystemResource(
                                "src/com/nt/cfgs/applicationContext.xml"));

        User1 user1 = (User1) factory.getBean("user1");
        System.out.println("user1");
    }
```

Bean Inheritance

⇒ We use Inheritance for reusability and extensibility. Bean Inheritance is nowhere related with classes level physical inheritance, It is all about XML file level bean configurations inheritance.

⇒ When we want to configure same class for multiple times as multiple spring beans we can reuse bean properties configurations across the multiple bean cfgs by using bean inheritance.

Eg.

Bike.java

```
package com.nt.beans;
public class Bike{
   private string id;
   private string make;
   private string enginecc;
  public void setId(string id){
        this.id=id;
       }
  public void setMake(string make){
        this.make=make;
       }
  public void setEnginecc(string enginecc){
        this.enginecc = enginecc;
       }
   //toString();
   ---
}
```

applicationContext.xml

```
<bean id="pulsor1" class="com.nt.beans.Bike">
 <property name="id" value="TS 08 1111"/>
 <property name="make" value="Bajaj"/>
 <property name="enginecc" value="150"/>
</bean>
<bean id="pulsor2" class="com.nt.beans.Bike">
<property name="id" value="TS 09 2222"/>
<property name="make" value="Bajaj"/>
<property name="enginecc" value="150"/>
</bean>
```

Problem: make, enginecc property values cfg is not having reusability even though they are same in multiple bean cfgs.

solution 1: (Bean Inheritance)

applicationContext.xml

```
<bean id="pulsor1" class="com.nt.beans.Bike">
    <property name="id" value="TS08 1111" />
    <property name="make" value="Bajaj" />
    <property name="enginecc" value="150" />
</bean>

<bean id="pulsor2" class="com.nt.beans.Bike" parent="pulsor1" />
    <property name="id" value="TS 08 2222" />
</bean>
```

solution 2: (Improved solution 1)

```
<bean id="basePulsor" class="com.nt.beans.Bike" abstract="true">
    <property name="make" value="Bajaj" />
    <property name="enginecc" value="150" />
</bean>
```
→ base Bean configuration

```
<bean id="pulsor1" class="com.nt.beans.Bike" parent="basePulsor">
    <property name="id" value="TS 08 1111" />
</bean>
```
→ child Bean configuration

```
<bean id="pulsor2" class="com.nt.beans.Bike" parent="basePulsor">
    <property name="id" value="TS 08 2222" />
</bean>
```
→ child Bean configuration

Imp points

⇒ It is not classes level physical inheritance.

⇒ Parent Bean cfg and child Bean cfg can refer same classes or different classes.

⇒ When abstract="true" it does not make bean class as abstract class but makes the bean cfg as the abstract bean cfg i.e, we can not call factory.getBean(-) having that abstract bean cfg bean id. (It throws Exception in bean creation).

8/6/15

Collection Merging

⇒ It always to merge the values set to Base Bean configuration collection property with the values set to the child bean cfg collection property by using merge="true" attribute.

EnggCourse.java

```
public class EnggCourse {
    private List<String> subjects;
    public void setSubjects(List<String> subjects) {
        this.subjects = subjects;
    }
    toString();
    --
}
```

Approach(1):
applicationContext.xml

```
<bean id="base1stYear" class="com.nt.beans.EnggCourse" abstract="true">
    <property name="subjects">
        <list>
            <value>C</value>
            <value>Maths</value>
        </list>
    </property>
</bean>

<bean id="ECE1stYear" class="com.nt.beans.EnggCourse" parent="base1stYear">
    <property name="subjects">
        <list>
            <value>Digital Electronics</value>
            <value>C</value>
            <value>Maths</value>
        </list>
    </property>
</bean>
```

Problem: Writing common subjects (C, Maths) again in child bean cfg indicates duplication.

Method(2)

applicationContext.xml

```
<bean id="ECE1stYear" class="com.nt.beans.EnggCourse" parent="baseIstYear">
  <property name="subjects">
    <list>
      <value> Digital Electronics </value>
    </list>
  </property>
</bean>
```

Problem: child bean cfg overrides the values of parent bean cfg.

Solution

applicationContext.xml

```
<bean id="ECE1stYear" class="com.nt.beans.EnggCourse" parent="baseIstYear">
  <property name="subjects">
    <list merge="true">
      <value> Digital Electronics </value>
    </list>
  </property>
</bean>
```

ClientApp.java

```
EnggCourse eceEngg = factory.getBean("ECE1stYear", EnggCourse.class);
S.o.p(eceEngg);
```

* Bean Aliasing

⇒ Alias name means nickname or petname. It is all about providing more alternate names/ids to use bean.

⇒ Prior to spring 2.0 "name" attribute is given to provide alias name.
from spring 2.0 <alias> tag is introduced for the same.

⇒ We can use both name attribute, <alias> tag to provide alias names from spring2.0

Eg.

```
<bean id="system" name="pc,desktop" class="com.nt.beans.Computer">
  ---
</bean>

<alias name="system" alias="machine" />
<alias name="system" alias="box" />
```

Note: <alias> tag is recomanded to used becoz it allows to provide alias name to

multiple bean in single place.

Note: Bean aliasing is useful in the maintanence mode of Bean cfgs where we can short alias names for lengthy bean id's.

In ClientApp

```
factory.getBean("box");
factory.getBean("pc");
factory.getBean("system");
```

comment: Performing dependency lookup by using IOC container

⇒ If target class (or) main class writes the code to search and gather dependent values (or objects is called "dependency lookup" (or) dependency pull.

⇒ By creating IOC container in target class (or) mainclass we can perform dependency lookup (or) pull operation to get dependent objects.

23/06/15

⇒ If the target class wants to use dependent object in all methods then go for dependency injection

⇒ If the target class wants to use dependent class object only in specific method then go for dependency lookup (or) pull.

Example

```
package com.nt.beans;
public class Engine{
    private String enggId;
    public void setEnggId(String enggId){
        this.enggId = enggId;
    }
    public void start(){
        S.o.p("Engine started");
    }
}
```

Vehicle.java

```
package com.nt.beans;
public class Vehicle{
    //injects Dependent class bean id.
    public void move(){
        BeanFactory factory = new XmlBeanFactory
        (new FileSystemResource("src/com/nt/cfgs/
                                 applicationContext.xml"));
        Engine engine = factory.getBean(enggId, Engine.class);
                                //dependency lookup
        engine.start();
        S.o.p("vehicle moved for journey");
    }
}
```

place this here

IOC Proj8 (Dependency Lookup - IDREF)

Engine

```
package com.nt.beans;

    public class Engine() {

          public Engine(){
             S.O.P(" Engine:0-param constructor");
            }
     public void start(){
           S.O.P("Engg started");
        }
     }

public class Vehicle{
     private string enggId;
     public Vehicle(){
         S.O.P("Vehicle: 0-param constructor");
       }
     public class void setEnggId( string enggId){
               this.enggId = enggId;
            }
     public void move(){

  BeanFactory factory =new XmlBeanFactory(new FileSystemResource("com/nt/cfgs/
                                                    applicationContext.xml"));
       Engine engine = factory.getBean(enggId, Engine.class);
          engine.start();
         S.O.P("Vehicle is moved for journey");
        }
}
```

applicationContext.xml

```
<bean id="enggi" class="com.nt.beans.Engine" />

<bean id="vehicle" class="com.nt.beans.Vehicle" >
    <property name="enggId" >
     <idref bean="enggi" />
    </property>
</bean>
```

ClientApp

```
public class ClientApp {
   public static void main(String[] args) {
   BeanFactory factory = new XmlBeanFactory( new FileSystemResource("src/com/nt/cfgs/
                                                    applicationContext.xml"));
      Vehicle vehicle = factory.getBean("vehicle", Vehicle.class);
         vehicle.move();
      }
}
```

package com.nt.beans.cfg;

applicationContext.xml

```
<bean id="vehicle" class="com.nt.beans.Vehicle">
   <property name="enggId" value="engg" /> //bean id is injected
  </bean>
  <bean id="engg" class="com.nt.beans.Engine" />
```

package test

ClientApp

In main(-) method:

```
BeanFactory factory = new XmlBeanFactory(new FileSystemResource("src/com/nt/cfg/
                                                    applicationContext.xml"));

    Vehicle vehicle = factory.getBean("vehicle", Vehicle.class);
      vehicle.move();
```

⇒In the above code,

```
<property name="enggId" value="engg" />
```

↳ Injects dependent class bean Id (Engine) to target class obj (Vehicle) as an ordinary string value with out checking whether bean class is configured or not with that bean Id. To overcome this problem and we inject beanId only when bean class is configured with bean Id we need to use '<idref>' tag.

Eg:(1)
```
<bean id = "vehicle" class = "com.nt.beans.Vehicle">
   <property name = "enggId">
     <idref bean = "engg" />
   </property>
  </bean>
  <bean id="engg" class = "com.nt.beans.Engine" />
```

Note: string name = "Test" → "Test" is maintained as string content.

Class name = Test.class → "Test" class is maintained as class in the object of java.lang.class;

Factory method

⇒The method that is capable of creating either same class object or different class object is called "factory method".

⇒ Two types of factory methods a) Instance Factory Method.

Eg(1) String s1 = new String("hello");

String s2 = s1.concat("123"); //hello123

↳ returns same class object.

Eg(2) StringBuffer sb = new StringBuffer("hello how are u");

String s1 = sb.substring(0,4); // gives hello

↳ returns different class object.

b) Static Factory Method

Eg Class c = Class.forName("java.util.Date");

↳ never creates object just loads the class.

↳ returns same class object.

Eg Console cons = System.console(); // creates console object

(final class) ↳ returns different class object.

Note: Calender never return its object because it is an abstract class. It always returns its subclass object.

⇒ We make IOC container to create Spring Bean class objects in 4 ways.

a) Using 0-param constructor
b) Using parameterized constructor
c) Using Instance factory Method
d) Using static factory Method.

* Difference b/t IOC & Dependency Injection (technical)

⇒ IOC is a specification including set of rules for doing collaboration & injections. Depend

eg: Injection is an implementation of doing collaborations & injections.

1) What is bean wiring ?
2) What is auto wiring ?

24/6/15

applicationContext.xml

```
<!-- using static factory method -->
<bean id="c1" class="java.lang.Class" factory-method="forName">
  <constructor-arg value="java.util.Date"/>
</bean> //uses Class.forName(-)
<!--here constructor-arg tag supplies arg values of factory Method -->
<bean id="c2" class="java.util.Calender" factory-method="getInstance"/>
  //uses Calender.getInstance()
```

```
<!--Using instance factory Methods-->
  <bean id="s1" class="java.lang.String">
    <constructor-arg value="hello" />
  </bean>
 <bean id="s2" factory-bean="s1" factory-method="concat">
   <constructor-arg value="how are u?" />
 </bean> //s1.concat(--)
 <bean id="sb" class="java.lang.StringBuffer">
   <constructor-arg value="Hi buddy" />
 </bean>
 <bean id="s3" factory-bean="sb" factory-method="substring">
    <constructor-arg value="0" />
    <constructor-arg value="2" />
 </bean> //uses sb.substring(0,2)
```

Client App

```
//getBean
Class c1=factory.getBean("c1", class.class);
S.O.P("c1 data ::: "+c1+" c1 class name ::: "+c1.getClass());
//getBean
Calender c2=factory.getBean("c2", Calender.class);
S.O.P("c2 data :::: "+c2+" c2 class name :::: "+c2.getClass());
//getBean
String s2=factory.getBean("s2", String.class);
S.O.P("s2 data :::: "+s2+" s2 class name :::: "+s2.getClass());
//getBean
String s3=factory.getBean("s3", String.class);
S.O.P("s3 data::"+s3+" s3 class name :::: "+s3.getClass());
```

Singleton class

→ The class which allows us to create single object per JVM is "Singleton class".

Reasons for Singleton class creation

1) When class is not having any state only behaviours (only methods but not variables)

2) If class having only readonly sharable state (all are final variables).

3) If class is allowing write operations in a synchronised manner. (having huge data to read & performing write operations on variables) ie, 1 thread at a time.

⇒ Instead of creating multiple objs with same data or no data, create 1 obj and use it for multiple times.

Singleton behaviour

⇒ If the class allows us to create multiple objects but we are having only one bean object creation and using this object for multiple times.

Eg: our servlet class is not singleton class because it allows to create multiple objects but most of the time servlet container creates only one object.

Bean Scopes

⇒ We can make IOC container keeping our spring bean class objects in different scopes.

They are,

In spring 1.x a) Singleton b) Prototype

In spring 2.x a) Singleton b) Prototype c) session d) request e) GlobalSession.

Note: globalSession is removed in spring 3.x and 4.x versions.

⇒ Default scope is "Singleton".

Singleton Scope

⇒ Returns same bean class object for all factory.getBean(-) method calls with same bean id.

Prototype Scope

⇒ Returns seperate bean class object for every factory.getBean() method call.

request

⇒ Bean class obj is specific to each request

session

⇒ Bean class obj is visible through out the session.

globalsession

⇒ Specific to portlets environment.

* Use "scope" attribute in <bean> tag to specify bean scope.

Note: request, session scopes can be used only in spring mvc applications.

Example

applicationContext.xml

```
<bean id="v1" class="com.nt.beans.Vehicle" scope="prototype">
---
</bean>
```

Client App

```
Vehicle v1=factory.getBean("v1", vehicle.class);
Vehicle v2 = factory.getBean("v1", vehicle.class);
S.O.P( v1.hashCode()+"  "+v2.hashCode());
S.O.P(v1==v2);
```

Note: The "bean scope singleton" never makes our bean class as "singleton Java class" but gives singleton behaviour by while creating object.

25/6/15

Bean Wiring

⇒ Configuring beans and their dependency injection in bean configuration file is called Wiring.

1) Explicit Wiring : Here we use <property>, <constructor-arg> tags for dependency injection configurations

2) Auto Wiring : Here the IOC container automatically detects dependents and injects them to target objs. Here no <property>, <constructor-args> tags are required.

Auto wiring/Implicit Wiring Limitations

a) Can be used only to inject objects but not the simple values

b) If IOC container multiple dependencies to inject them ambiguity problem may raise

c) Kill the readability of the xml file, so the bug fixing becomes very complex.

Note : Use auto wiring only in POC projects or PILOT project for Faster/Rapid application development.

⇒ By using "autowire" attribute of <bean> we can enable autowiring.

⇒ There are 4 modes of performing autowiring

a) byName b) byType c) constructor d) autodetect (Removed from spring 3.x onwards).

1) autowire = "byName"

⇒ Performs setter injection

⇒ For this Target class property name and Dependent class bean id must match.

⇒ Here there is no possibility of getting ambiguity problem.

TravelAgent.java (Target class)

```
public class TravelAgent {
    private TourPlan tp;
    public TravelAgent(){
        System.out.println("0-param constructor of TravelAgent");
    }
    public TravelAgent(TourPlan tp){
        this.tp=tp;
        System.out.println("1-param constructor of TravelAgent");
    }
    public void setTp(Tourplan tp){
        this.tp=tp;
        System.out.println("setter Method of TravelAgent");
    }
    //toString()
}
```

TourPlan.java (Dependent class)

```
public class TourPlan{
    private String[] places;
    //Setter Method
    //toString()
}
```

applicationContext.xml

```
<bean id="tp" class="com.nt.beans.Tourplan">
  <property name="places"> <!--explicit injection --> Wiring -->
    <list>
      <value> Delhi </value>
      <value> Simla </value>
    </list>
  </property>
</bean>

<bean id="tagent" class="com.nt.beans.TravelAgent" autowire="byName" />
                                                    <!--Autowiring -->
```

ClientApp

```
BeanFactory factory = new XmlBeanFactory(new FileSystemResource("src\com\

// GetBean
TravelAgent agent = factory.getBean("tagent",TravelAgent.class);
    System.out.println(agent);
```

2) autowire = "byType"

⇒ Performs setter Injection

⇒ For this target class property type and dependent class must match or must be compatible with each other.

⇒ There is a possibility of getting ambiguity problem.

eg 1:

```
<bean id="tplan" class="com.nt.beans.Tourplan">
---
</bean>
<bean id="tagent" class="com.nt.beans.TravelAgent" autowire="byType" />
```

Note If we configure "Tourplan" class one more time in spring bean configuration file with different "bean id" then we get Ambiguity problem.

3) autowire = "constructor"

⇒ Uses parameterized constructor to perform constructor injection.

⇒ There is possibility of getting ambiguity problem if multiple dependencies are found but it does not throw exception if target class contains 0-param constructor otherwise exception will be thrown.

Eg

```
<bean id="tp1" class="com.nt.beans.TourPlan">
---
</bean>

<bean id="tagent" class="com.nt.beans.TravelAgent" autowire="constructor"/>
```

Note If we configure "TourPlan" class one more time in spring bean configuration file with different "bean id" then we get ambiguity problem.

4) autowire = "autodetect" (try in spring 2.x)

⇒ If target class contains 0-param constructor then it performs "byType" mode autowiring

⇒ If target class contains parameterized constructor then it performs "constructor" mode autowiring.

⇒ Removed from spring 3.0 onwards

Eg

```
<bean id="tp1" class="com.nt.beans.Tourplan">
---
</bean>
<bean id="tagent" class="com.nt.beans.TravelAgent" autowire="autodetect">
```

5) autowire = "no"

⇒ Disables autowiring.

26/6/15

1) What happens if we enable both autowiring and explicit wiring on same bean property?

Ans) If both wirings are performing the same setter injection or the same constructor injection then explicit wiring values will be injected as the final values.

⇒ If both wirings are performing two different injections then the setter injection values will be injected as the final values.

dependency-check

⇒ While performing constructor injection by using certain parameterized constructor all the parameters of that constructor must be configured for wiring, if not configured exception will be raised.

⇒ The above restriction is not there for setter injection configuration. To enable such restrictions while working with setter injection go for dependency-check by adding "dependency-check" attribute in <bean> tag.

a) dependency-check = "simple" : checks whether all simple properties are configured for setter injection or not. If not configured Exception will be raised

b) dependency-check = "objects" : checks whether all reference type properties are for setter injection or not.

c) dependency-check = "all" : checks whether both simple and object type properties are configured for setter injection or not.

d) dependency-check = "none" : disables the dependencies check.

⇒ use spring-beans-2.0.xsd in applicationContext.xml

Example

Vehicle.java

```
package
public class Vehicle{
private int id;
private Engine engine;
 //setter Methods;
 // to string;
 }
```

Engine.java

```
package
public class Engine{
  private String type;
  //setter Method
  // toString()
  --
 }
```

applicationContext.xml

```
<bean id="eng" class="com.nt.beans.Engine">
<property name="type" value="diesel" >
</bean>
<bean id="vehicle" class=com.nt.beans.Vehicle"
                    dependency-check = "all" >
<property name="id" value="1001" />
<Property name="engine" ref="eng" />
</bean>
```

clientApp.java

```
// create IOC container
Vehicle v=factory.getBean("vehicle", Vehicle.class);
S.O.P(v);
```

Note : dependency-check " is removed from spring 2.5 because it allow to apply restriction on group of properties not on specific properties.
@Required is given as an alternate.

Nested Beanfactories

⇒ In Real time project contains multiple layers. Instead of configuring all the beans of these multiple layers in single IOC container. It is recomanded to take multiple containers on 1 per layer base.

⇒ To use the beans of IOC container in another IOC container it is recomanded to make them as Nested Container like Parent container and child container.

⇒ The beans of parent container can be used in child container beans but reverse is not possible.

⇒ Generally Service layer beans want to use the beans of persistance layer so we can cfg beans of persistence layer in Parent container and the beans of service layer in child container.

IOC Proj12 (Nested Beanfactories)

```
|--> src
     |--> com.nt.beans
     |       |--> BankLoanApprover.java
     |       |--> LoanDetails.java
     |--> com.nt.cfgs
     |       |--> approver-beans.xml
     |       |--> loan-beans.xml
     |--> test
             |--> ClientApp.java
```

```
package com.nt.beans;
public class LoanDetails
{
   private  int id;
   private  String loanType;
   private  String customer;

    //setter Methods() and getter Methods()
    //toString
}
```

```
package com.nt.beans;
public class BankLoanApprover {
  // private String status;
  private LoanDetails details;
  //setter Method()
  //toString()
  public String approveLoan(){
    if(details.getLoanType().equals("two-wheeler")){
      return "Loan is approved";
    }
    else{
      return "Loan is rejected";
    }
  }
}
```

package com.nt.cfgs

a) loan-beans.xml

```
<bean id="ld" class="com.nt.beans.LoanDetails">          <!--ld=loandetails-->
  <property name="loanId" value="111"/>
  <property name="loanType" value="two-wheeler"/>
  <property name="customer" value="raja"/>
</bean>
```

b) approver-beans.xml

```
<bean id="bla" class="com.nt.beans.BankLoanApprover">
  <Property name="details" ref="ld"/>          <!--bla=bankloanapprover-->
</bean>
```

ClientApp

```
//create parent Ioc container (Parent Beanfactory)
BeanFactory pfactory = new XmlBeanFactory(new FileSystemResource("
                                    src/com/nt/cfgs/loan-beans.xml"));
//create child Ioc container (child Beanfactory)
BeanFactory cfactory = new XmlBeanFactory(
                                    (new FileSystemResource("src/com/nt/cfgs/approval-beans.xml"), pfactory);
//Get Bean
BankLoanApprover approver = cfactory.getBean("bla", BankLoanApprover.class);
    s.o.p(approver);
```

27/6/15

⇒ While working with Nested Bean Factory / Nested containers the spring Bean cfg file associated with child container is called child cfg file and the file associated with parent container is called parent cfg file.

Can u explain various attributes of <ref> tag?

1) <ref local = "<bean id">" > : Locates given bean id based spring bean cfg from current/local spring bean cfg file (child cfg file).

eg: <property name = "details">
<ref local = "id" />
</property>

2) <ref parent = "<bean id>"> : Locates given bean id based spring bean cfg from the parent bean cfg file.

eg: <property name = "details">
<ref parent = "id" />
</property>

3) <ref bean = "<bean id">" > : First tries to locate from current/local bean cfg file if not available then it tries to locate from parent bean cfg file.

Eg: <property name = "details">
<ref bean = "id" />
</property>

<property name = "details" ref = "id" /> is equal to <ref bean = "id" />

P-Namespace & C-Namespace

⇒ In xml schema, namespace is a library contains set of xml tags. Every namespace is identified with its namespace uri.

⇒ In spring 3.0, p-namespace is introduced to perform setter injection configuration with out using the lengthy <property> tags. It allows to place attribute directly in <bean> tags having the following syntax.

p:<property name> = "value"
p:<property name>-ref = "<bean id>"

p-namespace uri is : http://www.springframework.org/schema/p

⇒ Similarly c-namespace are introduced from spring 3.1.1 version for constructor injection tags cfgs by avoiding the lengthy <constructor-arg> tags.

It allows to place attributes directly in <bean> tag having the following syntax.

c: <property name> = "value"

c: <property name> - ref = "<bean id>"

c-namespace url is : http://www.springframework.org/schema/c
(collect from spring reference

(For example example ref. page no: 62 & 63. Application: 18)

Working with Application Context container (It is also IOC container)

⇒ Beanfactory container means it is the IOC container that is created by creating Beanfactory object.

⇒ Application context container means it is the IOC container that is created by creating Applicationcontext object.

⇒ Application Context obj is the object of a java class that implements org.sf.context.ApplicationContext (Interface). (It is a sub interface of Beanfactory (interface)).

There are 3 popular implementation classes for Application context (Interface) using which we can create Applicationcontext container (IOC container).

1) FileSystemXmlApplicationcontext

⇒ creates ApplicationContext container by locating spring bean cfg file from the specified path of the file system.

Eg: ApplicationContext ctx = new FileSystemXmlApplicationContext("src/nit/cfg/ac.xml");

2) ClassPathXmlApplication context

⇒ creates Application context container by locating spring bean cfg file from the directories and jar files added to classpath.

Eg: ApplicationContext ctx = new ClassPathXmlApplicationContext("applicationContext.xml");

3)

3) XmlWebApplicationContext

⇒ creates Application context container in web applications.

2/9/15

Application Context Container

⇒ ApplicationContext container can perform all the operations of BeanFactory container and can also perform this additional operations.

a) Pre-Instantiation of singleton spring Beans
b) Ability to work with Properties file
c) Support for i18n (Internationalization)
d) Support for Event handling/Processing and etc..

a) Pre-Instantiation of singleton spring Beans

⇒ While working with BeanFactory container the container instantiates spring bean only when factory.getBean(-) method is called (for any scope bean).

⇒ While working with ApplicationContext container the singleton scope beans will instantiate the moment container is started. This is called pre-instantiation of singleton beans.

⇒ If prototype scope bean is dependent to singleton scope then prototype scope will also be pre-instantiated to support dependency injection on singleton scope bean. But it does not mean prototype scope bean has changed its scope to "singleton".

⇒ Bean pre-instantiation is very much similar to enabling <load-on-startup> on servlet.

Q We have spring bean cfg file with 10 beans. How can we enable pre-instantiation only on 5 beans?

Ans: Take 5 beans as singleton scope beans and other 5 beans as prototype beans and donot make prototype scope beans as dependent to singleton scope beans.

⇒ The Java class that contains persistence logic is called "DAO" (Data Access Object).

⇒ The Java class that contains business logic is called "Business Service/Service class."

⇒ Generally service class ~~method~~ need DAO class object.

⇒ Jdbc DataSource object represents jdbc con pool and we can use this Jdbc DataSource object to get con object from jdbc con pool.

⇒ Spring api gives "org.sf.jdbc.dataSource.DriverManagerDataSource" class to give DataSource object pointing to the IOC container managed jdbc con pool. It creates the con objs in conpool based on bean property values cfg. They are driverClassName, url, username, password.

Example Application

IOC Proj14-(Injecting DataSource)
- src
  - com.nt.cfgs
    - applicationContext.xml
  - com.nt.dao
    - LoanDAO.java
  - com.nt.service
    - LoanService.java

Table

customer-LoanInfo

cno (number)	cname (varchar)	pamt (float)	intramt (float)

↳ com.nt.bo
↳ CustomerBO.java
↳ test
↳ clientApp.java

jar files : IOC Lib jars + ojdbc14/6.jar + spring-jdbc-<version> + spring-tx-<version>.jar

⇒ The Java class with standards (minimum getter & setter Methods) is called "Java Bean".

⇒ In Real time the data b/w two components (or) layers (or) resources will not be sent as individual values, they will be placed in java bean class object and they will be sent to source layer to destination layer.

⇒ The Java bean whose object holds the enduser supplied input values (form data) is called "vo class (value object class)".

⇒ The java bean whose object holds the data required for persistence logic or data generated by the persistence logic is called "BO class (Business Object class)".

⇒ The Java bean whose object holds the data to transfer from 1 layer to another layer is called "DTO class". This class generally implements "Serializable Interface".

applicationContext.xml

```
<bean id="ppc" class="org.sf.beans.factory.config.PropertyPlaceholderConfigurator" >
<property name="location" value="com/nt/commons/DBdetails.properties"/></bean>
<bean id="drds" class="org.sf.jdbc.datasource.DriverManagerDataSource">
 <property name="driverClassName" value="${jdbc.driver}"/>
 <property name="url" value="${jdbc.url}"/>
 <property name="username" value="${db.user}"/>
 <property name="password" value="${db.pwd}"/>
</bean>
<bean id="dao" class="com.nt.dao.LoanDAO">
   <property name="ds" ref="drds"/>
</bean>
<bean id="service" class="com.nt.service.LoanService">
  <property name=dao ref="dao"/>
```

⇒ Generally the DAO class contains two parts.

a) Query part (String constants hold queries).
b) Code part (contains persistence logic).

30/6/15

(b) Working with properties file

⇒ The text file that maintains the entries as key value pairs is called "properties file".

⇒ We generally use properties file to maintain jdbc properties like Driver class Name, url, username and password. To make them as flexible to modify with out touching the source code.

⇒ Spring applications can achieve this flexibility through xml file (spring beancfg file)

If spring application is added as new Application to the existing project and if existing application that project are getting jdbc properties from properties file then we should make sure that spring application also gets jdbc properties from properties file.

⇒ In Application Context container environment., we need to configure "org.sf.beans.factory.config.PropertyPlaceHolderConfigurer" class as spring bean to make container to locate given properties file and to recognise place holders ${..} in spring bean cfg file. Each place holder represents 1 key of the properties file to get value from properties file.

DBDetails.properties (com/nt/commons)

```
jdbc.driver = oracle.jdbc.driver.OracleDriver
jdbc.url    = oracle:thin:@localhost:1521:xe
db.user     = scott
db.pass     = tiger
```

applicationContext.xml

```
<bean id="ppc" class="org.sf.beans.factory.config.PropertyPlaceHolderConfigurer">
    <property name="location" value="src/com/nt/commons/DBDetails.properties" />
</bean>

<bean id="dmds" class="org.sf.jdbc.dataSource.DriverManagerDataSource">
    <property name="driverClassName" value="${jdbc.driver}" />
    <property name="url" value="${jdbc.url}" />
    <property name="username" value="${jdbc.user}" />
    <property name="password" value="${jdbc.pass}" />
</bean>
```

↳ placeholder representing the key of the properties file.

Refer App12 of page NO: 56

11/7/15

(c) I18n (Internationalization)

⇒ Locale means country + language.

Eg: en-US
en-Br (English as it speaks in Britain)
fr-FR
hi-IN (Hindi as it speaks in India)

⇒ making our application working different locales is called enabling internationalization
It deals with a) displaying presentation labels.
b) displaying numbers
c) Currency symbols
d) Displaying Date values and etc..,

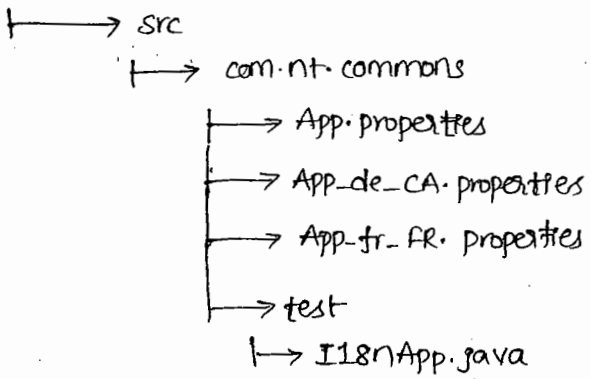
⇒ In core Java, JEE Applications we use ResourceBundle, Locale classes of java.util.package to work with I18n.

⇒ To apply I18n we need to take multiple properties files having presentation labels on 1 per locale basis. All these must have same keys and different values.

Eg: App.properties (base file)
App_de_DE.properties (for german)
and etc..

If no matching file is found the Base file will be used automatically.

Deployment Directory
I18nproject



Eg: Refer: Appl3 of page No: 57

⇒ We can use Google translater to get labels on other languages based on the given english words.

⇒ We can also place unicode Numbers in the properties files to render non english character labels.
Eg: Gothi gopi.com ⇒ unicode-editor (third party tool) and jdk's "native2ascii" tool.

Procedure

1) Download unicode editor from www.higopi.com website and type hindi words.
2) copy hindi words from unicode editor and place them in Notepad (save the file by taking "unicode" as the encoding type)

inputs.txt (hindi words)

मिटाना

सुरक्षित,

रोको

रद्द

3) use native2Ascii tool to get unicode numbers based on given input file.

>native2Ascii encoding = unicode inputs.txt output.txt
↳(contains unicode numbers).

output.txt (unicodes)

\u0930\u0926\u094d\u0926

\u0938\u0941\u0930\u0915\u094d\u0937\u093f\u0924,

\u0930\u094b\u0915\u094b

\u0930\u0926\u094d\u0926

Limitations in JEE/JSE environment while working with I18n.

a) In a web application if multiple servlet program/jsp program wants to display labels collected from properties then we need to create "ResourceBundle" class object in multiple servlet/jsp program or we need to place "ResaureBundle" obj as request attribute value.

NOTE : Some plan has to be decided to use ResourceBundle obj

b) If we want to display values of same key belonging to multiple locale property file then we need to create multiple ResourceBundle objects.

⇒ We can overcome these problems in Spring environment, by Application context container with the support of "ResourceBundleMessageSource" class.

Application on I18n (Internationalization) plain

App.properties
(#Basefile → English)
str1 = delete
str2 = save
str3 = stop
str4 = cancel

App.fr_CA.properties
#for french language
str1 = EFFACER
str2 = SAUF
str3 = ARRET
str4 = ANNULER

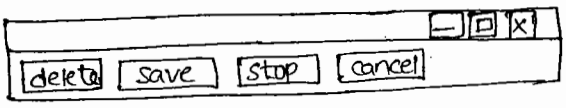
App.de_DE.properties
(#for german language)
str1 = LOSCHEN
str2 = DENN
str3 = ZUG
str4 = ABSAGEN

App.hi-IN.properties
(#for hindi language)
str1 = मिटाना
str2 = सुरक्षित
str3 = रोको
str4 = रद्द

I18nApp.java

```
//Internationalization of an application (the captions on buttons will come based on the
//locale specific properties file that is activated).

import java.awt.*;
import javax.swing.*;
import java.util.*;

public class I18nApp{
   public static void main(String args[]){
   //create Locale obj based on given language, country codes
   Locale l = new Locale (arg[0], arg[1]);

  //picks up properties file based on the Locale obj data
  ResourceBundle r = ResourceBundle.getBundle ("App", l);

    JFrame jf = new JFrame();
    Container cp = jf.getContentPane();
  //creates buttons by getting labels from activated properties file..
    JButton b1 = new JButton(r.getString("str1"));
    JButton b2 = new JButton(r.getString("str2"));
    JButton b3 = new JButton(r.getString("str3"));
    JButton b4 = new JButton(r.getString("str4"));

     cp.setLayout(new FlowLayout());
     cp.add(b1);
     cp.add(b2);
     cp.add(b3);
     cp.add(b4);

     jf.pack();
     jf.setVisible(true);

     }//main

}//class
```

```
>javac I18nApp.java
>java I18nApp fr CA  (App_fr_CA.properties file will be actived)
>java I18nApp hi IN  (App_hi_IN.properties file will be activated)
>java I18nApp x y  (App.properties file will be activated (base file))
```



2/7/15

I18n in Spring

⇒ Only ApplicationContext container supports I18n

⇒ we can use "ResourceBundleMessageSource" class as spring Bean by specifying base property file name as the value of "baseName" property.

⇒ Since we configure the above class as singleton scope spring bean the container creates only 1 object and uses it for multiple times.. but the above bean must be cfg with fixed bean id "messageSource".

⇒ we can call ctx.getMessage() to read messages from properties file.

⇒ In ApplicationContext, by supplying appropriate Locale object. This ctx.getMessage() method make container to locate properties file based on Locale obj data with the support of bean whose id "messageSource".

Advantages

⇒ By just creating 1 object for "ResourceBundleMessageSource" we can use properties files in multiple locations

⇒ Allows to read same message in different languages with one object of above class.

Example

IOC Proj15(I18n App)

```
|→ src
    |→ com.nt.commons
        |→ App.properties, App_fr_CA.properties
        |→ App_fr_FR.properties, App_hi_IN.properties
    |→ test
        |→ clientApp.java
    |→ com.nt.cfgs
        |→ applicationContext.xml
```

Refer the App 14 of page No:58459

applicationContext.xml

```
<beans>
<bean id="messageSource" class="org.sp.context.support.ResourceBundleMessageSource">
<property name="basenames">
<list>
    <value>App</value>
    <value>App_de_DE</value>
    <value>App_fr_CA</value>
</list>
</property> </bean> </beans>
```

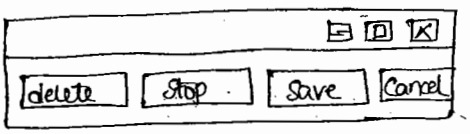
(fixed bean id.)

(only this filename is enough)

DemoClient.java

```
package com.nt.bean;
import org.springframework.context.support.*;
import java.util.*;
import javax.swing.*;
import java.awt.*;

public class DemoClient{
    public static void main (String[] args) throws Exception{
    //Locale Object
    Locale l=new Locale(args[0], args[1]);
  //Activate ApplicationContext container
ClassPathXmlApplicationContext ctx =new ClassPathXmlApplicationContext ("applicationContext.xml");
    //get Msgs from properties file
String msg1 = ctx.getMessage ("str1", null, "default msg1", l);   // str1 = key, default message if no message is found
String msg2 = ctx.getMessage ("str2", null, "default msg2", l);
String msg3 = ctx.getMessage ("str3", null, "default msg3", l);
String msg4 = ctx.getMessage ("str4", null, "default msg4", l);
                                          // object array to supply argument values
    //Develope Swing frame
    JFrame jf =new JFrame();
    Container cp = jf.getContentPane();
  //create buttons by getting label from activated properties file.
    JButton b1 = new JButton (msg1);
    JButton b2 = new JButton (msg2);
    JButton b3 = new JButton (msg3);
    JButton b4 = new JButton (msg4);
  cp.setLayout(new FlowLayout());
   cp.add(b1);
   cp.add(b2);
   cp.add(b3);
   cp.add(b4);
    jf.pack();
    jf.show();
  }//main
}//class
```

```
>javac DemoClient.java
>java DemoClient fr FR
>java DemoClient de DE
>java DemoClient x y
```



Event Handling

⇒ Event is an object/action that is raised on the component/object.

⇒ Event Handling means executing logic when event is raised, For this we need the support of EventListeners.

⇒ They are 4 actors in Event Processing /handling.

a) source object (eg: Button)

b) Event class (eg: ActionEvent)

c) Event Listener (eg: ActionListener)

d) Event handling method (eg: actionPerformed(-))

⇒ We can perform event handling on ApplicationContext container to keep tracic of when container started and stopped. This allows programmer to evaluate the performance of spring code/logics.

⇒ In spring Environment, for event handling.

a) Source object : ApplicationContext container

b) Event class : ApplicationEvent

c) Event listener : ApplicationListener

d) Event-handling method : onApplicationEvent(-)

↳ Executes when container is started and stopped.

Example App

1) Any Application developed using ApplicationContext containers has to be taken

2) Develop java class for event handling

3)

```
package com.nt.event;

public class IOCPMListener implements ApplicationListener {

    private long endtime;
    private long starttime;

    @Override
    public void onApplicationEvent(ApplicationEvent ae) {

        if(ae.toString().indexOf("Refreshed") != -1) {
        } starttime = System.currentTimeMillis();
```

```
else
{
endtime = System.currentTimeMillis();
System.out.println("Container Active time " + (endtime-starttime) + "ms");
}
}// onApplicationEvent()
}// class
```

3) Cfg the above listener class in applicationContext container

```
<bean class="com.nt.event.IOCMListener"/>
```

4) create ApplicationContext container and stop that in the client Application.

ClientApp.java.

```
FileSystemXmlApplicationContext ctx = new FileSystemXmlApplicationContext("src/com/nt/cfgs/
                                      applicationContext.xml");
                                  --> // starts the container
---
---
---
ctx.close(); closes the container.
```

Note: ctx.getBean(-) / factory.getBean(-) gives spring Bean class object based on the given bean id.

Note: ctx.getMessage(-,-,-,-) gives local specific message collected based on from properties based on the given "key".

3/7/15

Where to use BeanFactory? Where to use ApplicationContext?

⇒ If we are developing small scale applns like Mobile Apps, Embedded system Apps having the suppot of spring then use BeanFactory (Here few extra KBs also matters towards appln size).

⇒ If we are developing Enterprise applns (like web applications, Distributed Applns) then use ApplicationContext (always prefer working with ApplicationContext).

Similarities and differences b/w BeanFactory and ApplicationContext containers?

Operation	BeanFactory	ApplicationContext
1) Bean Instantiation	yes	yes
2) Bean Wiring	yes	yes
3) Auto Wiring	yes	yes
4) Properties file support	Directly Not possible	yes
5) I18n	no	yes
6) Event processing	no	yes
7) Automatic Registration of Bean PostProcessor	no	yes
8) Automatic Registration of BeanFactory PostProcessor	no	yes
9) stopping/closing container	no	yes

Bean Life cycle

⇒ It talks about spring Bean class object creation to Destruction.

⇒ In servlet Life cycle we use the following Life cycle methods.

a) init(-) b) service(-,-) c) destroy()

⇒ In ordinary class we use constructor, finalize() methods to place initialization and uninitialization logics.

⇒ Spring Bean allows two life cycle methods.

a) init(): Executes after all dependency injections. so we can used to verify injects are done properly (or) not.

b) destroy(): Executes when IOC container is about to destroy our bean class object. Useful to place uninitialization logics.

⇒ In Spring we can perform Bean Life cycle operations in 3 ways.

a) Declarative Approach

b) Programatic Approach

c) ~~Declarative Approach~~ Annotation Approach

a) Declarative Approach

⇒ Allows to configure two user-defined methods as bean life cycle methods. These methods are called customInit method, customdestroy method. These methods return type should be void and must not contain params.

⇒ We must configure the above methods as life cycle methods using init-method, destroy-method attributes of <bean> tag.

Example :

IOC Proj16 (Bean Life Cycle)

- com.nt.beans
  - checkVoting.java
- com.nt.cfgs
  - applicationContext.xml
- test
  - clientApp.java

NOTE: dependency check is removed from spring 2.x version onwards.

⇒ In custom init method we generally donot write initialization because it wont execute after bean instantiation. It executes after completing all injections.

⇒ While working with ApplicationContext container when we close the container by calling ctx.close(). The container destroys bean class objects. When this task is about to happen custom destroy is execute.

⇒ While working with BeanFactory container we can use factory.destroySingleton() to destroy all singleton scope bean class objects.

Limitations with Declarative approach of Bean Life Cycle

1) If u forget to configure life cycle methods in xml file they will not be executed.
2) Identifying life cycle methods is very complex while dealing with predefined, third party supplied classes as spring bean.

THIS

(b) Programmatic Approach

⇒ Here our bean class must implement InitializingBean(Interface) and DisposableBean(Interface) given by Spring API (Bean class becomes spring api dependent) - (Invasive).

⇒ InitializingBean(Interface) gives afterPropertiesSet(), DisposableBean(Interface) gives destroy() method.

⇒ we use afterPropertiesSet() method as alternate to customInit() method and destroy() method as alternate to custom destroy() method.

Sample Code

```
public class CheckVoting implements InitializingBean, DisposableBean {
    private int age;
    private String name, address;
    //setters
    ---
    ---
    public void afterPropertiesSet(){
        if (age<=0 || name=null) throw new IllegalArgumentException("name,
                                                    age must set");
    }
    public void destroy(){
        age=0;
        name=null;
        address=null;
    }
    ---
    ---
}
```

Note: There is no need to cfg any life cycle methods in xml file during spring bean cfg.

Q: If we place Bean life cycle configurations in both programatic approach and Declarative Approach then what happen?

⇒ custom-init method executes after InitializingBean's afterPropertiesSet() method.
⇒ custom-destroy method executes after DisposableBean's destroy() method.

Note It is recomanded to use 1 approach Bean life cycle at a time.

Note Programatic / ~~Declarative~~ Approach makes our spring bean class as spring api dependent. Most of spring api supplied classes are given based on Programatic approach of Bean Life cycle.

Note ⇒ Custom init() method and InitializingBean's afterPropertiesSet() is useful to check whether important bean properties are injected with values or not.

⇒ Custom destroy() method, DisposableBean's destroy() is useful to nullify bean properties or to release non-java resources that are associated with Bean.

(c) Annotations Approach

Refer Spring Core Annotations Section.

Conclusion on Bean Life Cycle :

⇒ While working with user-defined spring beans prefer Declarative approach or Annotation Approach

⇒ While working with Third party api supplied classes as spring beans use Declarative Approach.

⇒ While working with Spring api supplied classes as spring beans, first check for Programatic approach implementation if not found then go for Declarative Approach.

06/7/15

Aware Injection / Interface Injection

1) If underlying container assigns or injects object/value to target class only when target class implements/extends certain interface/class is called contextual dependency lookup.

2) Sewlet container assigns configured object to our sewlet class only when our sewlet class implements java.sewlet.Sewlet(Interface) directly or indirectly (contextual dependency lookup).

3) If one class need another class then we go for either dependency lookup (or) dependency injection

4) If target class wants to use dependent object in all the methods then go for dependency injection. If target class wants to dependent object in specific methods then go for dependency lookup.

5) If we perform dependency lookup in regular fashion we need to create one more IOC container in target class to get Dependent class object (refer 23rd June). So instead of creating separate IOC container in target class we can ask underlying IOC container (Client App started) to inject BeanFactory or Application Context object to target class by Implementing BeanFactoryAware or ApplicationContextAware interfaces on target class and we can use that object to get dependent class object.

6) Here underlying container is injecting object value/ only when target class implements XxxAware interface so it is called AwareInjection or Interface Injection or Contextual Dependency Lookup (CDL).

The XxxAware Interfaces are

BeanNameAware → To inject current bean object

BeanFactoryAware → To inject BeanFactory object

ApplicationContextAware → To inject ApplicationContext object.

NOTE

The IOC container is, created by creating BeanFactory object internally maintains bean ids, BeanFactory obj. The IOC container that created by creating ApplicationContext Object & etc....

All these objects can be injected to any class through AwareInjection.

Example

IOCProg17(AwareInjection or) InterfaceInjection (or) contextual dependency Lookup).

```
|--> src
     |--> com.nt.cfgs
          |--> applicationContext.xml
     |--> com.nt.beans
          |--> Vehicle.java
          |--> Engine.java
     |--> test
          |--> ClientApp.java.
```

applicationContext.xml

```
<beans>
  <bean id="engg" class="com.nt.beans.Engine"/>
  <bean id="vehicle" class="com.nt.beans.Vehicle">
    <property name="enggId" value="engg"/>
  </bean>
</beans>
```

Engine.java

```
package com.nt.beans;
  public class Engine
  {  public Engine(){
       System.out.println("0-param: Engine");
     }
     public void start(){
       System.out.println("Engine started");
     }
  }
```

Eg(1) Vehicle.java

```
package com.nt.beans;

import org.springframework.beans.BeansException;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.context.ApplicationContext;
import org.springframework.context.ApplicationContextAware;
import org.springframework.core.io.FileSystemResource;

public class Vehicle implements ApplicationContextAware {
        private String enggId;
        private ApplicationContext ctx;
    public void setEnggId(String enggId) {
            S.O.P("Vehicle : setEngineId");
            this.enggId = enggId;
        }
    public vehicle() {
        S.O.P("0-param: vehicle");
      }
    public void move() {
        Engine engg = ctx.getBean(enggId, Engine.class);
           engg.start();
          S.O.P("Vehicle moved for journey");
        }
    @Override
    public void setApplicationContext(ApplicationContext ctx) throws BeansException {
            S.O.P("Vehicle: setApplicationContext");
            this.ctx = ctx;
           }
    }
```

Eg(2) Vehicle.java.

Vehicle (BeanFactoryAware, BeanNameAware)

```
public class Vehicle implements BeanFactoryAware, BeanNameAware {
            private String enggId;
            private ApplicationContext ctx;
        public void setEnggId(String enggId) {
                    S.O.P("Vehicle : setEngineId");
                this.enggId = enggId; }
```

```
public vehicle(){
    S.O.P("0-param: vehicle");
  }
  public void move(){
      Engine engg = ctx.getBean(enggId, Engine.class);
        engg.start();
      S.O.P("vehicle moved for journey");
    }
@Override
public void setApplicationContext(ApplicationContext ctx) throws BeansException {
      S.O.P("vehicle: setApplicationContext");
    this.ctx = ctx;
  }
@Override
  public void setBeanName(String beanName){
      S.O.P("Bean ID:" + beanName);
  }
@Override  BeanFactory factory;
  public void setBeanFactory(BeanFactory factory) throws BeansException{
      this.factory = factory;
      S.O.P("vehicle: setBean:" + factory);
    }
}
```

clientApp.Java

```
public class clientApp{
    public static void main(String[] args){
ApplicationContext ctx = new FileSystemXmlApplicationContext("src/com/int/cfgs/applicationContext.xml");
    Vehicle vehicle = ctx.getBean("vehicle", Vehicle.class);
      vehicle.move();
    }
}
```

o/p (1) normal

```
0-param: Engine
0-param: vehicle
vehicle: setEngineId
vehicle: setApplicationContext
Engine started
vehicle moved for journey
```

o/p (2) BeanFactoryAware, BeanNameAware,

```
0-param: Engine
0-param: vehicle
vehicle: setEngineId
BeanID: vehicle
vehicle: setBean: org.springframework.beans.
  support.DefaultListableBeanFactory@2c8d66f2:
vehicle: setApplicationContext
Engine started
Vehicle moved for started journey.
```

FactoryBean

1) The class which, capable of creating and returning objects is called "factory class".
2) In spring factoryBean is a selfless bean that means it does not give its own class object, it always gives one Resultant object.
3) To develop class as FactoryBean class, the class must implement the spring api's FactoryBean(Interface) & should provide definitions for 3 methods.
   a) getObject() → creates & return resultant object
   b) getObjectType() → return java.lang.class object holding resultant object classname
   c) isSingleton() → specifies weather Resultant obj should be taken as singleton obj (for true) or prototype object (for false).
4) The Bean that gets object from JNDI Registry based on given JNDI name will be developed as FactoryBean.

Example

IOC Prog18(FactoryBean)
- src
  - com.nt.cfgs
    - applicationContext.xml
  - com.nt.beans
    - ScheduleReminder.java, DateFactoryBean.java
  - test
    - ClientApp.java

applicationContext.xml

```
<beans>
   <bean id="dfb" class="com.nt.beans.DateFactoryBean">
         <constructor-arg value="110"/>
         <constructor-arg value="10"/>
         <constructor-arg value="20"/>
   </bean>
   <bean id="reminder" class="com.nt.beans.ScheduleReminder">
         <property name="date" ref="dfb"/>
   </bean>
</beans>
```

ScheduleReminder.java

```
package com.nt.beans;
import java.util.Date;
public class ScheduleReminder{
    private Date date;
    public void setDate(Date date){
        this.date = date;
    }
    @Override
    public String toString(){
        return "ScheduleReminder [date="+date+"]";
    }
}
```

DateFactoryBean.java

```
package com.nt.beans;
import java.util.Date;
import org.springframework.beans.factory.FactoryBean;
public class DateFactoryBean implements FactoryBean<Date>{
    private int year, month, date;
    public DateFactoryBean(int year, int month, int date){
        this.year = year;
        this.month = month;
        this.date = date;
    }
    @Override
    public Date getObject() throws Exception{
        S.O.P("DFB: getObject()");
        return new Date(year, month, date);
    }
    @Override
    public Class<?> getObjectType(){
        S.O.P("DFB: getObjectType()");
        return Date.class;
    }
```

```
    @Override
        public boolean isSingleton(){
            S.O.P("DFB: isSingleton()");
            return true;
        }
    }
```

client App.java

```
public class ClientApp{
        public static void main(String[] args){
        //create IOC container
    ApplicationContext ctx=new FileSystemXmlApplicationContext("src\com\nt\cfgs\
                                                    applicationcontext.xml");
        //getBean
    ScheduleReminder reminder=ctx.getBean("reminder",ScheduleReminder.class);
        System.out.println(reminder);
    }
}
```

o/p

```
DFB : isSingleton()
DFB : getObject()
DFB : isSingleton()
DFB : getObjectType()
ScheduleReminder [date = Sat Nov 20 00:00:00 IST 2010]
```

How FactoryBean is able to give another obj (Resultant object)?

IOC container create FactoryBean class object normally based on the configurations done in spring bean configuration file, but before keeping that object in a scope IOC container checks whether the class of object is implementing FactoryBean(Interface or not). If implementing it calls getObject() to get Resultant object and calls isSingleton() to keep Resultant obj as singleton scope or prototype scope object with the bean id given for factoryBean configuration (xml file).

07/07/15

Method Replacement

⇒ In normal classes if we want to replace existing logics of business method with new logics. We have two options

a) Comment the existing logic and write new logics in the same business method.

b) Develope sub class and override business method with new logics.

⇒ In the above model we need to modify source code to revert back original logics if we add new logics on experimental basic or if new logics execution is failed in production environment at client side.

⇒ The another problem with above approach is reverting back to original business logic is complex when source code (.java) is not exposed.

⇒ In Retail industry offers will be placed for certain period, once the offers is over at then standard prices will be charged. To execute new logic during offer period and to revert back to original logics once the offer is over with out touching source code we can not use above methods.

⇒ To overcome the above problem use Method Replacer/Replacement concept of spring-for this we need to use separate class implementing Method Replacer (Interface) and we need to write new logics in reimplement(–,–,–).

public Object reimplement(Object obj, Method method, object[] args).

Object obj → Gives the access to original bean class object

Method method → gives the access to original method that we want to replace

object[] args → gives access original method args

⇒ We need to cfg method replace class as spring bean and we need to IOC container to execute reimplement(–,–,–) when original method is called, for this we need to use

<replaced-method name="↑originalmethod" replacer="↑Bean id of method replacer class." />

For example App refer page No: 65 & 66.

Eg 1

```
<bean id="banic" class="com.nt.beans.Bank">
    <replaced-method method="calcIntrAmt" replace="mr" />
</bean>
<bean id="mr" class="com.nt.beans.IntrRateReplace" />
```

If overloaded methods are there and if we want to perform method replacement only for specific method then use <arg-type> tags as shown below.

```
<bean id="bank"  class="com.nt.beans.Bank">
  <replaced-method  method="calcIntAmt" replace="mr">
    <arg-type> float </arg-type>
   <arg-type > float </arg-type>
   </replaced-method>
  </bean>
  ---
  ----
```

Internals

⇒ IOC container creates a subclass for main bean class (Bank) like Bank$CGLIB by using CGLIB Libraries internally. In this class business method (calcIntrAmt()) will be overridden. In this method definition the object of MethodReplacer class (IntrRateReplace) will be accessed and reimplement method will be called due to this new logic will be executed. When client Application calls ctx.getBean() or factory.getBean, the above generated subclass object will be returned.

Sample class

```
public class Bank$CGLIB extends Bank implements ApplicationContextAware {
  private ApplicationContext ctx;
  public setApplicationContext (ApplicationContext ctx){
      this.ctx=ctx;
   }
  public float calcIntAmt(-,-,-){
     IntRateReplacer ir= ctx.getBean("mr", IntRateReplacer.class");
     float retVal = ir.reimplement(-,-,-);
    return retval;
   }
}
```

Important points

⇒ The method that you want to replace should not be final, static (this method can not be overridden in subclass).

⇒ Eventhough we can write replacement logic for multiple methods in single reimplement method by using if conditions, it is recommanded to take separate method replacer for every class.

IOCProj19 (Method Replacer)

com.nt.cfgs

```
<beans>
  <bean id="bank" class="com.nt.beans.Bank">
    <replaced-method name="calIntrAmt" replacer="mr">
        <arg-type> float </arg-type>
        <arg-type> float </arg-type>
    </replaced-method>
  </bean>
  <bean id="mr" class="com.nt.beans.IntrRateReplacer" />
</beans>
```

com.nt.beans

Bank.java

```
package com.nt.beans;

public class Bank{
    public float calcIntrAmt(float pamt, float time) {
        S.O.P("bank: calcIntrAmt(-,--,)");
        return pamt*time*2.0f/100;
    }
    public float calcIntrAmt(){
        return 123.0f;
    }
}
```

IntrRateReplacer.java

```
import java.lang.reflect.Method;
import java.util.Arrays;
import org.sf.beans.factory.support.MethodReplacer;

public class IntrRateReplacer implements MethodReplacer{
    @Override
    public Object reimplement(Object bean, Method method, Object[] args) throws Throwable{
        S.O.P("IntrRateReplacer: reimplement(-,-,-,-)");
```

```
        S.O.P(" bean class: "+ bean.getClass());
        S.O.P(" method name: " + method.getName());
        S.O.P(" method args: " + Arrays.toString(args));
      If(method.getName().equals("calcIntrAmt")
        {
            float pamt = (Float) args[0];
            float time = (Float) args[1];
            return pamt * time * 1.2/100.0f;
        }
      else{
            return 0.0f;
          }
    }
}
```

test

client App.java

```
import com.nt.beans.Bank;
public class clientApp{
      public static void main(String[] args){
ApplicationContext ctx = new FileSystemApplicationContext("src/com/nt/cfgs/applicationContext.xml");
          Bank bank = ctx.getBean("bank", Bank.class);
      S.O.P("Intr Amt:" + bank.calcIntrAmt(1000,20));
      }
}
```

op

```
IntrRateReplacer : reimplement(-,-,-,-)
bean class : class com.nt.beans.Bank$$EnhancedByCGLIB $$8224cb30
method class: calcIntrAmt
method args : [1000.0, 20.0]
Intr Amt: 2400.0
```

09/07/15

Q) Explain various injections supported by spring?

a) Setter injection
b) Constructor injection
c) Aware Injection / Context Dependency Lookup / Interface injection
d) Method injection (Method Replacer)
e) Lookup method Injection

Lookup Method Injection

Problem with dependency Injection

Problem: When prototype scope bean is dependent to singleton scope bean some how the prototype scope bean also acts as singleton scope bean.

Eg All request coming to web container (application) will be handled single Servlet-container web container (~~request~~) (singleton). This container uses separate RequestHandler to process each request so Servlet Container target class and RequestHandler is Dependent class, Hole over ServletContainer should be taken as singleton and scope bean and RequestHandler should be taken as prototype scope but because the above problem "RequestHandler" also acts as the singleton scope bean.

```
<bean id = "container" class = "com.nt.beans.MyServletContainer" scope = "singleton" >
    <property name = "handler" ref = "rh" />
</bean>
<bean id = "rh" class = "com.nt.beans.RequestHandler" scope = "prototype" />
```

Target Bean Scope	Dependent Bean Scope	Resultant scope of Dependent Bean
prototype	singleton	singleton (ok)
singleton	singleton	singleton (ok)
prototype	prototype	prototype (ok)
singleton	prototype	singleton (not ok) (use dependency Lookup method or lookup method injection)

10/07/15

⇒ We should not think about dependency injection (constructor / setter) in the following situations

a) If Dependent Bean class object is required in Target Bean only in specific methods
b) If Target Bean scope is singleton and Dependent Bean scope is Prototype (Refer above table).

In the above situation prefer using Dependency lookup,

a) If we go for traditional dependency lookup code we need to created an extra IOC container in the methods of target class. (Which is bad practise).

b) Using Aware injection we can avoid that extra IOC container towards dependency lookup operations. But it kills the non-invasive behaviour of spring Appln because target bean class should implement Awarexxx interfaces.

c) To overcome the above two problems we can use Lookup method injection (LMI) which says perform "dependency lookup" but container should write code for it (not the programmer).

To perform LMI we need to consider the following points

a) Target Bean class should have abstract method returning Dependent Bean class (for this target bean class must be taken as Abstract class)

b) we must configure the above abstract method along with targetbean class configured by using <lookup-method> tag.

Refer IOCProj 20, 21, 22(Final).

Note We can configure abstract class as spring bean only when <lookup-method> tag is placed for lookup method injection otherwise not possible.

Spring IOC container generates sub class for target Bean class and implements the abstract method that class to get and return dependent bean class object.

for this it internally uses CGLIB libraries.

⇒ In spring 2.x, 3.x we need to supply CGLIB libraries separately where as spring 4.x not required.

⇒ Lookup method injection usecases.

Sewlet container <--> RequestHandler
(singleton) (prototype)

Manager <--> Customer
(singleton) (Prototype)

LoanApprover <--> LoanDetails
(singleton) (prototype)

IOC Prog 20 (LM Injection Problem)

```
<bean id = "container" class = "com.nt.beans.MyServletContainer" scope = "singleton">
    <property name = "handler" ref = "rh" />
</bean>
<bean id = "rh" class = "com.nt.beans.RequestHandler" scope = "prototype" />
```

MyServletContainer.java

```
public class MyServletContainer {
    private RequestHandler handler;
    public MyServletContainer() {
        S.O.P ("MyServletContainer : 0-param constructor");
    }
    public void setHandler(RequestHandler handler) {
        this.handler = handler;
    }
    public void processRequest(String data) {
        handler.handle(data);
    }
}
```

RequestHandler.java

```
public class RequestHandler {
    private static int count;
    public RequestHandler() {
        count++;
        S.O.P("Request Handler : 0-param constructor");
    }
    public static void setCount(int count) {
        RequestHandler.count = count;
    }
    public void handle(String data)
    {
        S.O.P("HandleRequest:" + count + "with data " + data);
    }
}
```

ClientApp.java

```
ctx = new FileSystemXmlApplicationContext("src/com/nt/cfgs/applicationContext.xml");
MyServletContainer container = ctx.getBean("container", MyServletContainer.class);
container.processRequest("hello");
container.processRequest("hai");
```

O/P

```
MyServletContainer : 0-param constructor
RequestHandler : 0-param constructor
HandleRequest : 1 with data hello
HandleRequest : 1 with data hai
```

IOC Proj21 (LMInjection Solution)

```
<bean id = "container" class = "com.nt.beans.MyServletContainer" scope = "singleton" >
   <property name = "beanId" >
      <idref bean = "rh" />
   </property>
</bean>
<bean id = "rh" class = "com.nt.beans.RequestHandler" scope = "prototype" />
```

MyServletContainer.java

```
public class MyServletContainer implements ApplicationContextAware{
        private String beanId;
      public MyServletContainer(){
              s.o.p("MyServletContainer :0-param constructor");
        }
       public String getBeanId(){
            return beanId;
        }
      public void setBeanId(String beanId){
            this.beanId = beanId;
        }
   ApplicationContext ctx;
   @Override
  public void setApplicationContext(ApplicationContext ctx) throws BeanException{
          this.ctx=ctx;
      }
      public void processRequest(String data){
          RequestHandler handler = ctx.getBean("beanId", RequestHandler.class);
            handler.handle(data);
        }
}
```

RequestHandler.java

```
public class RequestHandler{
      private static int count;
        public RequestHandler(){
           count++;
         s.o.p("RequestHandler :0-param constructor");
        }
    public static int getCount(){
          return count;
        }
  public static void setCount(int count){
              RequestHandler.count =count; }
```

```
    public void handle(String data){
        S.O.P("HandleRequest : " + count + " with data " +data);
        }
}
```

ClientApp.java.

```
public class ClientApp{

   public static void main(String[] args){
ApplicationContext ctx = new FileSystemXmlApplicationContext("src\com\int\cfgs\
                                                    applicationContext.xml");
MyServletContainer container = ctx.getBean("container", MyServletContainer.class);
        container.processRequest("hello");
        container.processRequest("hai");
    }
}
```

o/p

```
MyServletContainer : 0-param constructor
Request Handler : 0-param constructor
HandleRequest : 1 with data hello
Request Handler : 0-param constructor
HandleRequest : 2 with data hai
```

BeanPostProcessor (BPP)

⇒ Assigning some values to bean properties after injection is called "Bean Post Processing"

⇒ We can do this post processing inside the bean class and specific to bean class by using custom-init method or afterPropertiesSet() of bean class.
↳ (Initializing Bean's)

⇒ If we want to apply common post processing logic for multiple beans that to being from outside the bean class then we can use BeanPostProcessor class.

⇒ The BeanPostProcessor class executes for all beans that are configure in spring bean configuration file.

⇒ To develop BeanPostProcessor we need to take a class that implements BeanPostProcessor (Interface) should provide implementation for two methods.

a) postProcessBeforeInitialization (Object bean, String beanId) Executes before afterProperties() for every bean class.

b) postProcessAfterInitialization (Object bean, String beanId) Executes after customInit() for every bean class.

⇒ This concept is useful to develop custom post process / initialization logic for all beans in one class.

IOCProg23 (BeanPostProcessor)

```
|--> src
     |--> com.nt.service
          |--> CustStudService.java
     |--> com.nt.dao
          |--> CustStudDAO.java
     |--> com.nt.bo
          |--> StudentBO.java
          |--> CustomerBO.java
          |--> BaseBO.java
     |--> com.nt.cfgs
          |--> applicationContext.xml
     |--> test
          |--> ClientApp.java
     |--> com.nt.bpp
          |--> BOPostProcessor.java
```

14/7/15

⇒ In realtime, bean post processor will be used to place common post process initialization logic that is requried for multiple beans like assigning sysdate as dateofjoing (or) last modified date as date of updation etc..

⇒ This concept is very useful towards auditing activity.

⇒ While working with application context container if we just configure BeanPostProcessor class in xml file as spring bean. The container automatically recognise and use that Bean post Processor.

⇒ While working with bean factory container we must register BeanPostProcessor explicitly.

```
BOPostProcessor bpp = new BOPostProcessor; // BPP class
((ConfigurableListableBeanFactory) factory).addBeanPostProcessor(bpp);
```

IOCProj23 (BPP)

BaseBO.java

```
public abstract class BaseBO {
    private int id;
    Private String name;
    private Date doj;
    public BaseBO{
        S.O.P("BaseBO: 0-param constructor");
        }
    //setters and getters methods
    }
```

StudentBO.java

```
public class studentBO extends BaseBO{
    private string course;
    public StudentBO(){
        S.O.P("StudentBO : 0-param constructor");
        }
    public string getCourse(){
        return course;
        }
    public void setCourse(string course){
        this.course = course;
        }
    }
```

CustomerBO.java

```
public class CustomerBO extends BaseBO{
    private float billAmt;
    public CustomerBO(){
        S.O.P("CustomerBO: 0-param constructor");
        }
    //setters and getters method
}
```

BOPostProcessor.java

```
public class BOPostProcessor implements BeanPostProcessor{
        public BOPostProcessor(){
          S.O.P("BPP: 0-param constructor");
        }
       @Override
    public Object postProcessBeforeInitialization(Object bean, String bid) throws BeanException{
        S.O.P("BPP: BeforeInitialization(-,-)");
        if(bean instanceof BaseBO){
            ((BaseBO)bean).setDoj(new Date());
            return bean;
          }
          return bean;
       }
     @Override
    public Object postProcessAfterInitialization(Object bean, String bid) throws BeanException{
          S.O.P("BPP: AfterInitialization(-,-)");
        return bean;
      }
}
```

CustStudService.java

```
public abstract class CustStudService{
     private CustStudDAO dao;
      public CustStudService(){
            S.O.P("Service: 0-param constructor");
      }
    public abstract StudentBO getStudentBO();
    public abstract CustomerBO getCustomerBO();
      public void setDao(CustStudDAO dao){
                  this.dao=dao;}
```

```
public void processStudent(int id, String name, String course){
        s.o.p("Processing Student Info");
    StudentBO stbo = getStudentBO();
        stbo.setId(id);
        stbo.setName(name);
        stbo.setCourse(course);
        //use dao
        dao.insertData(stbo);
    }
public void processCustomer(int id, String name, float billAmt){
        s.o.p("Processing Customer Info");
        CustomerBO custbo = getCustomerBO();
            custbo.setId(id);
            custbo.setName(name);
            custbo.setBillAmt(billAmt);
            //use dao
            dao.insertData(custbo);
        }
}
```

CustStudDAO.java

```
public class CustStudDAO{
        public CustStudDAO(){
            s.o.p("DAO:0-param constructor");
    }
 public void insertData(StudentBO stbo){
            s.o.p("Student Data is");
            s.o.p("stbo.getId()+" "+stbo.getName()+" "+stbo.getCourse()+" "+stbo.getDoj());
        }
public void insertData(CustomerBO custbo){
        s.o.p("Customer Data is");
        s.o.p("custbo.getId()+" "+custbo.getName()+" "+custbo.getBillAmt()+" "+
                                                    custbo.getDoj());
    }
}
```

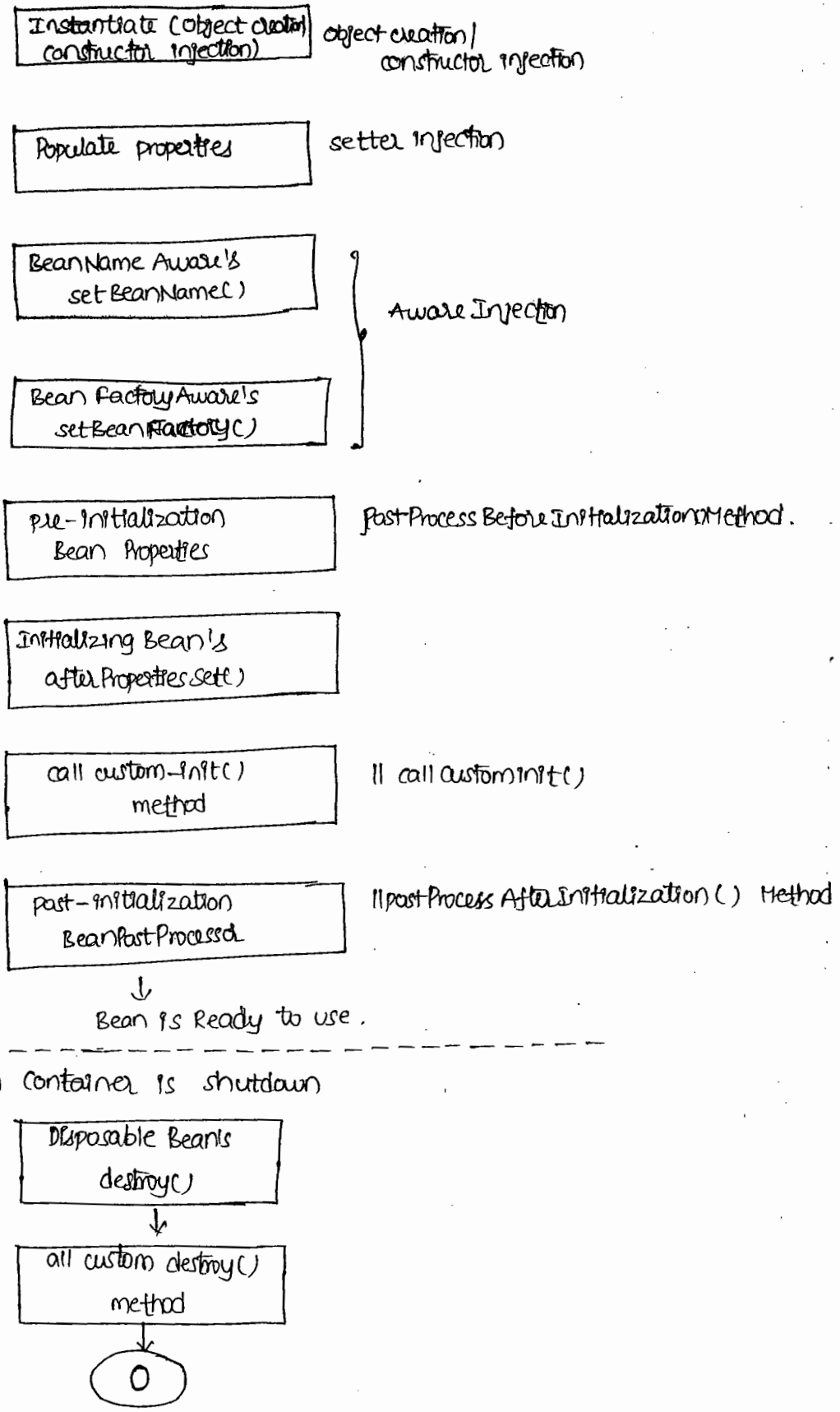
applicationContext.xml

```
<bean id="stBO" class="com.nt.bo.StudentBO" scope="prototype" />
<bean id="custBO" class="com.nt.bo.CustomerBO" scope="prototype" />
<bean id="dao" class="com.nt.dao.CustStudDAO" scope="singleton" />
<bean id="service" class="com.nt.service.CustStudService" scope="singleton" />
    <property name="dao" ref=dao" />
    <lookup-method name="getStudentBO" bean="stBO" />
    <lookup-method name="getCustomerBO" bean="custBO" />
</bean>
<bean id="bpp" class="com.nt.bpp.BOPostProcessor" />
```

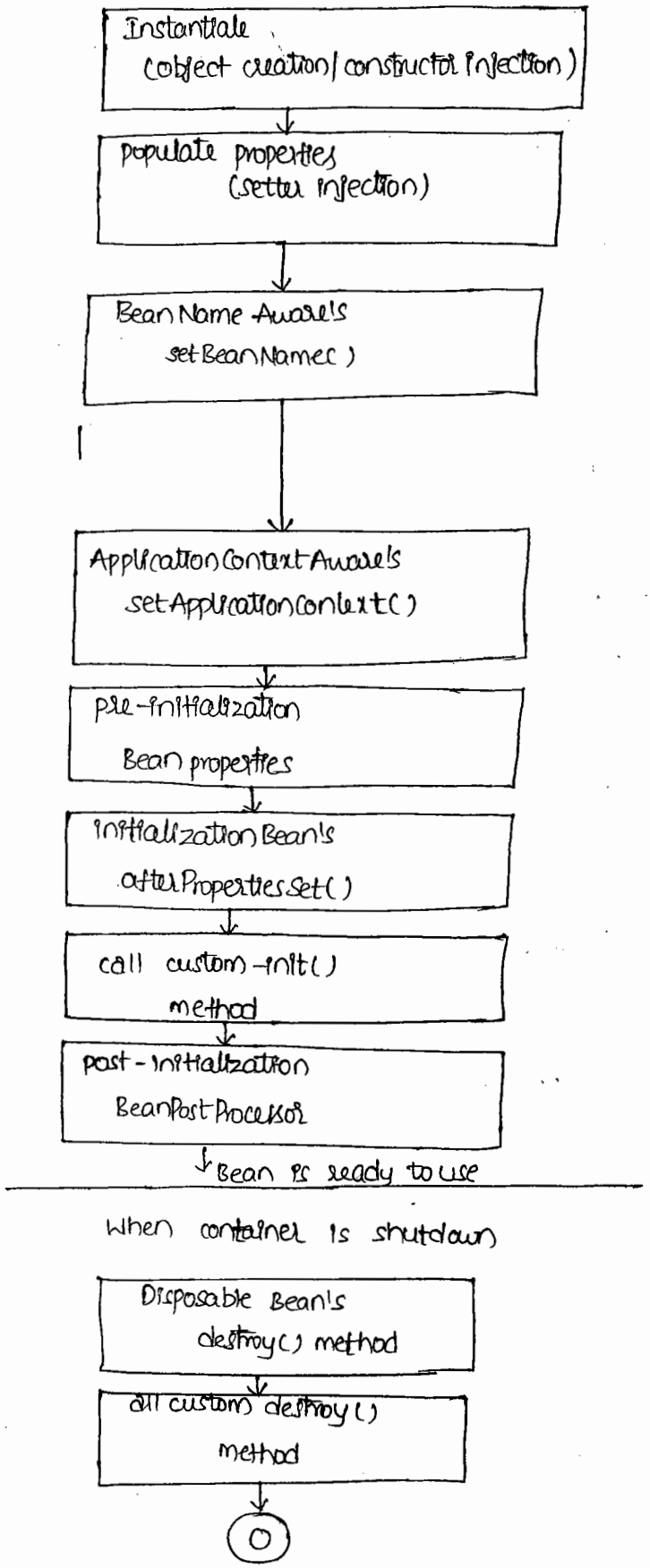
ClientApp.java

```
public class ClientApp{
    public class void main(String[] args){
        //create IOC container
  ApplicationContext ctx = new FileSystemXmlApplicationContext("/applicationContext.xml");
  /* //register BeanPostProcessor
     BOPostProcessor bpp = new BOPostProcessor();
  ((ConfigurableListableBeanFactory) factory).addBeanPostProcessor(bpp); */
        //get Bean
    CustStudService service = ctx.getBean("service", CustStudService.class);
        //call B.methods
      service.processCustomer(1001, "raja", 9000);
      service.processStudent(101, "kalan", "java");

      service.processCustomer(1002, "raja", 9000);
      service.processStudent(102, "kasan", "java");
    }
}
```

Bean LifeCycle diagram w.r.t BeanFactory container (Page NO: 32 & 33)



Bean Life Cycle diagram w.r.t to ApplicationContext Container



IOC Proj24(Bean Life Cycle Diagram)

MyPostProcessor.java

```
public class MyPostProcessor implements BeanPostProcessor {
    @Override
    public Object postProcessAfterInitialization(Object bean, String id) throws
                                                        BeanException {
        S.O.P("BPP: postProcessAfterInitialization");
        return bean;
    }
    @Override
    public Object postProcessBeforeInitialization(Object bean, String id) throws BeanException{
        S.O.P("BPP: postProcessBeforeInitialization");
        return bean;
    }
}
```

SeviceBean.java

```
public class SeviceBean implements InitializingBean, DisposableBean, ApplicationContextAware,
                    BeanNameAware, BeanFactoryAware {
    private String msg;
    public SeviceBean(){
        S.O.P("SeviceBean: 0-param constructor");
    }
    public SeviceBean(String msg) {
        S.O.P("SeviceBean: 1-param constructor");
        this.msg =msg;
    }
    public void setMsg(String msg){
        this.msg =msg;
        S.O.P("SeviceBean: setMsg(-)");
    }
```

```
@Override
public void setBeanFactory(BeanFactory arg0) throws BeansException {
        s.o.p("SeviceBean : setBeanFactory(-)");
    }
@override
public void setBeanName(String arg0) {
        s.o.p("SeviceBean : setBeanName(-)");
    }
@override
public void destroy() throws Exception {
        s.o.p("SeviceBean : destroy()");
    }
@override
public void afterPropertiesSet() throws Exception {
        s.o.p("SeviceBean : afterPropertiesSet()");
        }
@override
public void setApplicationContext(ApplicationContext arg0) throws BeansException {
        s.o.p("Sevice : setApplicationContext(-)");
    }
Public void myInit() {
        s.o.p("SeviceBean : myInit()");
    }
public void myDestroy() {
        s.o.p("SeviceBean : myDestroy()");
    }
public String generateWish() {
        return "Good Morning:: + "msg;
    }
}
```

applicationContext.xml

```
<bean id="service" class="com.nt.beans.SeviceBean" init-method="myInit"
                                                    destroy-method="myDestroy">
    <constructor-arg value="hello" />
    <property name="msg" value="hello" />
  </bean>

<bean id="bpp" class="com.nt.beans.MyPostProcessor" />
```

ClientApp.java

```
public class ClientApp{
    public static void main(String[] args){
     //create IOC container
  ApplicationContext ctx = new FileSystemXmlApplicationContext ("/applicationContext.xml");
     //getBean
    /*SeviceBean sevice = ctx.getBean ("service", SeviceBean.class);
      S.O.P ("service.generateWish() ");*/
    ((FileSystemXmlApplicationContext)ctx).close();
    }
}
```

12/7/15

Complete internal flow of core module application

a) Programmer creates BeanFactory object

b) Given spring bean cfg file will be loaded and it will be verified whether document is well formed, valid document or not.

c) Creates in memory logical IOC container in JVM's memory

d) Reads spring bean cfg file entries and creates InMemoryMetaData having all the cfgs.

e) Client Appln calls factory.getBean("id") method.

f) Loads Bean class from InMemoryMetaData based on the given bean id and instantiates that Bean based on constructor injection cfgs.

g) Keep bean class obj in the specified scope (If cyclic Dependency is found BeanInstantiation fails)

h) Performs setter injection based on cfgs

i) Performs Aware injection based on the Awarexxx interfaces that are implemented

j) Calls postProcessorBeforeInitialization() if BeanPostProcessor is added

k) Calls afterPropertiesSet() if InitializationBean(Interface) implemented

l) Calls custom init() method if "init-method" attribute is specified

m) Calls postProcessAfterInitialization()

n) Returns BeanClassObject to client Appln.

Note

1) If ApplicationContext object is created points (f) to (n) above also takes place with out calling getBean() for singleton scope bean.

2) If <lookup-method> (or) <replaced-method> tags are cfg inside <bean> tag then container does not create object for given bean class. It creates object for the subclass of the bean class.

BeanFactory Post Processor:

1. BeanPostProcessor is useful to write post process initialization logic for multiple beans.
2. BeanFactoryPostProcessor is useful to add extra cfgs by modifying in memory metadata after the creation of IOC container & before accessing Bean object by calling getBean().
3. It acts as extension hook/anchor for IOC container to add extra cfgs with out touching xml files (Spring bean cfgs file) i.e., it modifies then in memory metadata of IOC container dynamically at runtime after IOC container creation before bean class object is accessed.
4. To develop custom BeanFactoryPostProcessor we need to take a class implementing BeanFactoryPostProcessor (interface).
5. Spring has given lots of pre-defined BeanFactoryPostProcessor like PropertyPlaceHolderConfigurer. This class modifies in memory meta data by replacing placeholders ${...} with the values collected from properties file
6. ApplicationContext object based IOC container automatically recognizes the BeanFactoryPostProcessor ie, cfg in xml file. But for BeanFactory Object based IOC container we need register explicitly.
7. By default the IOC container of "BeanFactory" cannot work with properties file and placeholder. But we can make it working by registering "PropertyPlaceHolderConfigurer" as BeanFactoryPostProcessor.

Ex (w.r.t DAO Example)

In applicationContext.xml

```
<bean id="ppc" class="org.sf.beans.factory.config.PropertyPlaceHolderConfigurer">
<property name="location" value="com/nt/commons/DBDetails.properties"/>
<\bean>
```

In ClientApp

```
BeanFactory factory = new xmlBeanFactory(.....);
  //get PropertyPlaceHolderConfigurer object (ppho)
BeanFactoryPostProcessor bfpp = factory.getBean("ppc", PropertyPlaceHolderConfigurer.class);
  // register
   bfpp.postProcessBeanFactory((ConfigurableListableBeanFactory)factory);
```

13/04/15

Property Editors

⇒ In spring we use setter or constructor injection to assign values to bean properties.

⇒ In spring bean cfg file we configure the dependent values of as string values but they will be converted into int, long, float, Date & etc. values as required for bean properties automatically. For this spring internally uses PropertyEditors to convert string values to appropriate values.

⇒ All property editors are the classes implementing java.beans.PropertyEditor(Interface) spring has given multiple-built-in property editors like StringArrayPropertyEditor, byteArrayPropertyEditors, DateFormatPropertyEditor & etc..

Understanding Built-in PropertyEditors

```
package com.nt.beans;
public class AdhaarDetails {

   private int uid;
   private String name;
   private Date dob;
   private File photo;
   private String[] verifiers;

   // setters
   ---
   // toString()
    ---

   }
```

applicationContext.xml

```
<bean id="adhaar" class="package.AdhaarDetails">
<property name="uid" value="101" />
<property name="name" value="raja" />
<property name="dob" value="12/11/1980" />(*1)
<property name="photo" value="c:/abc/pics.gif" />
<property name="verifiers" value="sachin, Ramesh, ravi" />
</bean>.
```

*1: Internally uses date format property editor to convert given date value to java.util.Date class object. It allows only specific pattern date.

NOTE: In both containers (BF, AC) built-in property editors are automatically registered.

ClientApp.java

```
Main()
{
  AC ctx = ---
  //getBean

  AdharDetails details = ctx.getBean("adhar", AdharDetails.class);
  S.o.p(*details);
}
```

Spring allows to create custom property editor class either by implementing java.beans.PropertyEditor(Interface) or by extending from PropertyEditorSupport(class) recommended.

IntrAmtDetails.java

```
public class IntrAmtDetails
{
    private float p,t,r;
    //setters
}
```

IntrAmtCalculator.java

```
public class IntrAmtCalculator
{ private IntrAmtDetails details;
  //setters
}
```

In applicationContext.xml

```
<bean id="amtDetails" class="prg.IntrAmtDetails">
<property name="p" value="1000" />
          ="t" value="10" />
          ="r" value="2" />
</bean>
<bean id="intrCalculator" class="prg.IntrAmtCalculator">
<property name="details" value= ref="amtDetails" />
</bean>
```

But we want to write like this

applicationContext.xml

```
<bean id="intrCalculator" class="prog.IntrAmtCalculator" scope="prototype">
<property name="details" value="1000,10,2"/>
</bean>
```

⇒ Here details is reference type property. But we want to perform same values based injection, which is not possible directly. So we must go for custom property editor development.

IOCProj26(CustomPropertyEditor)

```
|→ src
    |→ com.nt.cfgs
        |→ applicationContext.xml
    |→ com.nt.beans
        |→ IntrAmtDetails.java, IntrAmtCalculator.java
    |→ com.nt.pe
        |→ CustomIntrAmtEditor.java
|→ test
    |→ ClientApp.java
```

⇒ Every IOC container internally maintains 1 built-in object called PropertyEditorRegistry and IOC container expose that object to a class that implements PropertyEditorRegistry (Interface) as the arguments of "registerCustomEditors()" method.

⇒ so that we can use that object to register CustomPropertyEditor by specifying property type & object of CustomPropertyEditor.

⇒ PropertyEditorRegistry is an interface that makes IOC container to expose the internally maintained PropertyEditorRegistry object to its implementation class.

4/7/15

SPRING CORE MODULE WITH ANNOTATIONS

Annotations:

1) Data about data or code about code is called "Metadata". Annotations are java statements to perform metadata operations in java files.

2) Annotations are alternate from xml files to perform metadata, resource configuration operations.

3) xml files gives good flexibility of modification but gives bad performance because the xml parsers are heavy to process xml documents.

4) Annotations do not provide flexibility of modification but provides good performance.

Syntax: @<Annotation name> (param1 = val1, ...)

5) Annotations for documentation are introduced from jdk1.0 and annotations for programming are introduced from jdk1.5.

6) We can apply annotations at different levels like class level, method level, field level, param level & etc.

7) The Annotations will be used by compilers, containers, annotation processors, JRE & etc..

8) Some annotations provide instructions in the generation of api docs.

Ex: @authors, @since etc..

9) Some annotations provide instructions to compiler but no effect will be there at runtime.

Ex: @override (tells compiler to that it is inherited method of a class)

10) Some Annotations are useful to make the classes/interfaces as special items.

@Entity makes ordinary java bean as HB Domain class

@WebServlet makes class as servlet component.

11) Some Annotations are useful to generate new classes using tools dynamically based on given class code.

Ex: @WebService

12) All java technologies, fl w's that are designed based on jdk1.5 are supporting annotations.

Ex: servlet 3.x, HB 3.2+, Spring 2.0+, EJB 3.0+ & etc..

13) If we perform certain settings by using annotations & xml files then the settings done through annotations.

IOC Prog26 (Custom PropertyEditor)

IntrAmtDetails.java

```
public class IntrAmtDetails {

    private int principle;
    private int time;
    private int rate;

    public void setPrinciple(int principle){
        this.principle = principle;
    }
    public void setTime(int time){
        this.time =time;
    }
    public void setRate(int rate){
        this.rate =rate;
    }
    public int getPrinciple(){
        return principle;
    }
    public int getTime(){
        return time;
    }
    public int getRate(){
        return rate;
    }
}
```

IntrAmtCalculator.java

```
public class IntrAmtCalculator{

    private IntrAmtDetails details;

    public void setDetails(IntrAmtDetails details){
        this.details =details;
    }
    public float calcIntrAmt(){
        return (details.getPrinciple()*details.getRate()*details.getTime())/100.0f;
    }
}
```

applicationContext.xml

```
<bean id="intrCalculator" class="com.nt.beans.IntrAmtCalculator" scope="prototype">
    <property name="details" value="1000,20,2" />
</bean>

<!-- <bean id="amtDetails" class="com.nt.beans.IntrAmtDetails">
    <property name="principle" value="1000" />
    <property name="time" value="10" />
    <property name="rate" value="2" />
</bean>
<bean id="intrCalculator" class="com.nt.beans.IntrAmtCalculator">
    <property name="details" ref="amtDetails" />
</bean> -->
```

CustomIntrAmtEditor.java

```
public class CustomIntrAmtEditor extends PropertyEditorSupport {
    public CustomIntrAmtEditor(){
        S.O.P("CustomIntrAmtEditor : 0-param constructor");
    }
@Override
public void setAsText(String text) throws IllegalArgumentException {
    S.O.P("CustomIntrAmtEditor : setAsText(-)");
    //split the given value
    int principle = Integer.parseInt(text.substring(0,text.indexOf(",")));
    int time = Integer.parseInt(text.substring(text.indexOf(",")+1,text.lastIndexOf(",")));
    int rate = Integer.parseInt(text.substring(text.lastIndexOf(",")+1,text.length()));
    IntrAmtDetails details = new IntrAmtDetails();
    details.setPrinciple(principle);
    details.setTime(time);
    details.setRate(rate);
    setValue(details);
} //method
} //class
```

TestClient.java

```
public class TestClient {
    public static void main(String[] args) {
    //create IOC container
  BeanFactory factory = new XmlBeanFactory(new FileSystemResource("./applicationContext.xml"));
  ((ConfigurableListableBeanFactory)factory).addPropertyEditorRegistrar(new MyCustomPE());
   //get Bean
  IntrAmtCalculator ical = factory.getBean("intrCalculator", IntrAmtCalculator.class);
     S.O.P("Intr Amt "+ical.calc.calcIntrAmt());
  }//main
}//class
class MyCustomPE implements PropertyEditorRegistrar {
      @Override
   public void registerCustomEditors(PropertyEditorRegistry registry){
       registry.registerCustomEditor(IntrAmtDetails.class, new CustomIntrAmtEditor());
       }
}
```

14/7/15

SPRING CORE MODULE WITH ANNOTATIONS

Annotations

1) Data about data or code about code is called "Meta Data". Annotations are java statements to perform metadata operations in java files.

2) Annotations are alternate for xml files to perform metadata, resource configuration operations

3) xml files gives good flexibility of modification but gives bad performance because the xml parsers are heavy to process xml documents.

4) Annotations donot provide flexibility of modification but provides good performance.

Syntax: @<Annotation name>(param1=value1,---)

5) Annotations for documentation are introduced from jdk1.0 and annotations for programming are introduced from jdk1.5.

6) we can apply annotations at different levels like class level, method level, field level, param level etc.

7) The Annotations will be used by compilers, containers, annotation processors, JRE etc.

8) Some annotations provide instructions in the generation of api docs.

Ex: @author, @since etc.,

9) some annotations provide instructions to compiler but no effect will be there at runtime.

Ex: @override (tells compiler that it is inherited method of a class)

10) some annotations are useful to mark the classes/interfaces as special items.

@Entity makes ordinary java bean as HB Domain class

@WebServlet makes class as servlet component

11) Some annotations are useful to generate new classes using tools dynamically based on given class code

Ex: @Webservice

12) All java technologies, fwls that are designed based on jdk1.5 are supporting annotations

Ex: servlet 3.x, HB 3.2+, Spring 2.0+, EJB 3.0+ etc.

13) If we perform certain settings by using annotations & xml files then the settings done through annotations.

15/7/15

SPRING ANNOTATIONS (FOR CORE MODULE)

⇒ Spring 2.0 onwards there is a suppolt fol annotations and more annotations are added incremently in next versions.

⇒ In spring programming we need to use both xml and annotations together because we can instruct IOC container to recognize & use annotations only through xml file we cannot use annotations to make predefined classes as the spring beans because we cannot add annotations by opening the source code of those classes. To cfg such classes as spring beans we need to work with xml suppolt.

Spring 2.0

@ Configuration
@ Required
@ Repositoly

Spring 2.5

@ Autowired
@ Qualifier
@ Scope

Stereotyped annotations.

@ Component
@ Service
@ Controller

Spring 3.0

@ Bean
@ Dependson
@ Lazy
@ Value

Java config annotations (can be used only with spring 3.x)

@ Post Construct
@ PreDestroy
@ Resource
@ Inject
@ Named

@Required :

⇒ To make dependency injection mandatory on properties we can use dependency check activity. But it's removed from spring 2.5 because it doesn't allow the programmer to enable the restriction on specific property. It allows the programmer to enable restrictions on group of properties like "simple, object, all". To overcome this problem we can use @Required.

Root.java

```
public class Robot {
  private int id;
  private String type;
 @Required
  public void setId(int id)
  {
    this.id = id;
  }
  public void setType(String type)
  {
    this.type = type;
  }
  //toString()   @Override
  ---          public String toString(){
               return "Robot [id = "+id+", type = "+type+"] ";
}                    }
```

applicationContext.xml

```
<bean id="robot" class="prg.Robot">
 <property name="id" value="1001" />
<!--<property name="type" value="army" /> -->
 </bean>
<!--<bean class="org.sf.beans.factory.annotations.RequiredAnnotationBeanPostProcessor" /> -->
        (or)
 <context:annotation-config/>
```

ClientApp.java

```
//create IOC container
ApplicationContext ctx = new FileSystemXmlApplicationContext("/applicationContext.xml");
//getBean
Robot r=(Robot) ctx.getBean("robot");
S.O.P(r);
```

⇒ The BeanPostProcessor of every annotations contains logic to recognize the annotation and to provide functionality for the annotation. But configure the BeanPostProcessor of every annotation is complex process. To overcome this problem use <context:annotation-config/> (To work with this tag import schema namespace in the xml).

16-7-15

Autowiring by using Orbitary Method

```
public class Vehicle
{
  private Engine engg;
  @Autowired(required=true)
  public void assign(Engine engg)
  { this.engg=engg;
  }
  //toString()
  ---
}
```

⇒ Here assign is the orbitary method.

Autowiring by using Constructor

```
public class Vehicle
{
  private Engine engg;
  @Autowired(required=true)
  public Vehicle(Engine engg)
  { this.engg=engg;
  }
  //toString()
  ---
}
```

Engine.java

```
public class Engine{
    private int id;
    private String type;
  public int getId(){
      return id;
    }
  public void setId(int id){
      this.id=id;
    }
  public String getType(){
      return type;
    }
public void setType(String type){
      this.type=type;
    }
  //toString
}
```

⇒ We can't apply @Autowired on the top of 0-param constructor.

⇒ At max only 1 constructor can have @Autowired annotation.

⇒ When @Autowired is placed on the top of field (or) setter method (or) arbitary method then "byType" mode autowiring takes place. When the same annotation is placed on the top of constructor then constructor mode autowiring takes place.

@Qualifier

⇒ While performing "byType" mode autowiring there is a possibility of getting ambiguity problem if multiple dependents are found to inject to target class. In order to overcome this problem we need to use @Qualifier annotation in which IOC container to perform "byname" mode of autowiring.

Ex

```
package com.nt.beans;
public class Vehicle{
  @Autowired
  @Qualifier (name="eng")
  private Engine engg;
   //toString()
  --
}
```

application Context.xml

```
<bean id="eng" class="pkg.Engine" >
---      <property name="id" value="11"/> <property name="type" value="CRDI"/>
</bean>   <qualifier value="eng"/>

<bean id="eng1" class="pkg.Engine">
---  <property name="id" value="12"/><property name="type" value="DDIS"/>
</bean>  <qualifier value="e1"/>

<bean id="vehicle" class="pkg.Vehicle">

<context:annotation-config/>
```

ClientApp.java

```
//create IOC container
ApplicationContext ctx = new FileSystemXmlApplicationContext ("/applicationContext.xml");
  //getBean
Vehicle vehicle = ctx.getBean("vehicle", Vehicle.class);
   S.O.P(vehicle);
```

@Component, @Service, @Repository, @Controller

⇒ The above annotations are called as stereotype annotations because all the four annotations are having same purpose with minor changes.

⇒ We can use all the 4 annotations to configure java classes as the spring beans.

@Component : Generic purpose stereotype annotation to configure any java class as spring bean ie, exposes given java class as spring bean to IOC container.

@Service : Specialization of @Component, useful to configure the java classes of service layer. As of now no special behaviour so it is same as @Component.

@Repository : Specialization of @Component, useful to configure the DAO classes as the spring bean. Capable of translating one form of exceptions to another form of exceptions.

@Controller : Specialization of @Component, use to configure classes as the controller of spring mvc appln to process request given by clients.

@Value : It is given to configure dependent values for simple properties of dependency injection.

JAVA CONFING ANNOTATIONS:

@Component, @Autowired etc.. are spring supplied annotations. If we use those annotations the code of spring bean becomes tightly coupled code to spring fw. This kills non-invasive feature of spring. To overcome this problem use java config annotations given by JEE, JSE annotations. For these annotations functionalities will be given by the containers of fw that we are using.

Ex

@Inject annotation spring provides its functionality & EJB provides its functionality. so if our code moves from one fw to another fw we can still continue the same ole java config annotations.

Engine.java

```
@Component("eng")
public class Engine{
    @Value("1001")
    private int id;
    @Value("DDIS")
    private String type;

  //setters
 public void setId(int id){
        this.id=id;
       }
public void setType(String type){
        this.type=type;
        }
  //toString()
@Override
public String toString(){
  return "Engine [id="+id+", type="+type+"]";
   }
}
```

Vehicle.java

```
public class Vehicle{
    @Autowired
  private Engine engg;
    //toString()
   @Override
  public String toString(){
    return "vehicle [engg="+engg+"]";
     }
}
```

applicationContext.xml

```
<context:component-scan base-package="com.nt.beans"/>
```

ClientApp.java

```
public class ClientApp{
        public static void main(String[] args){
          //create IOC container
ApplicationContext ctx=new FileSystemXmlApplicationContext("/applicationContext.xml");
        //get Bean
       Vehicle  vh=ctx.getBean("vh",Vehicle.class);
         s.o.p(vh);
     }
}
```

17/7/15

These Annotations are part of JSE, JEE specifications but implementations will be provided in different levels and frameworks. so if change runtime libraries of Appln we can still use same annotations in the classes..

Eg @Inject, @Named, @Resource, @PostConstruct, @PreDestroy and etc..

@Inject

⇒ It is given by JEE specification in javax.inject package having ability of performing by Name|byType|constructor mode of autowiring.

⇒ This is similar to @Autowired but the differences are

a) no required param in @Inject

b) @Inject is a java config annotation so non-invasive spring application can be developed.

Vehicle.java.

```
package com.nt.beans;
  @Component("vehicle")
  public class Vehicle{
    @Inject
    Private Engine engine;

    //toString()
    ---
    }
```

@Inject can be applied at field level | method level | constructor level.
↳ setter | arbitary method.

IocProj0 (@Inject, @Named)

⇒ To work with @Inject, @Named kind of java config annotation we need to add javax.inject.jar file.

Total jar files : IOC lib + spring-aop-<version>.jar + javax.inject.jar

@Named.

⇒ It is java config annotation of javax.inject. package (JEE specification). This annotation can be used in two ways

a) To solve ambiguity problem that raised while performing "byType" mode of autowiring (alternate to @Qualifier). Can be used along with @Inject, @Resource, @Autowired.

b) To cfg/make java classes as spring beans as alternative to spring specific annotation like @Component and etc..

vehicle.java

```
package com.nt.beans;

@Named("vehicle") //bean configuration
public class Vehicle
{
  @Inject
  @Named("eng1") //byname mode autowiring
  private Engine engg;

//toString()    public void assign(Engine engg){
 ---                  this.engg=engg;
}                   }
```

ClientApp.java

```
Public class ClientApp{
  public static void main(String[] args){
ApplicationContext ctx=new FileSystemXmlApplicationContext("/applicationContext.xml");
  //get Bean
 Vehicle vehicle = ctx.getBean("vehicle", Vehicle.class);
    S.o.p(vehicle);
  }}
```

Engine.java

```
@Named("eng")
public class Engine{
   @Value("1001")
  private int id;
   @Value("DDIS")
 private String type;
  //setters
  --
//toString()
--
}
```

applicationContext.xml

```
<!--<bean id="eng1" class="com.nt.beans.Engine">
  ---<property name="id" value="1001"/> <property name="type" value="DDIS"/>
 </bean>
<bean id="eng2" class="com.nt.beans.Engine">
  <property name="id" value="1002"/> <property name="type" value="CRDI"/>
 </bean> -->
<context:component-scan base-package="com.nt.beans" />
```

@Resource

⇒ It is jdk supplied java config annotation of javax.annotation package.
⇒ can be used to perform byType, byName mode of autowiring.
⇒ can be applied at field / method (setter, arbitary) level.
⇒ By using name param of this annotation we can perform byName mode of autowiring.

Vehicle.java.

```
package com.rt.beans;
@Named("vehicle")
public class Vehicle{
@Resource // performs byType mode autowiring
private Engine engg;   public void assign(Engine engg){
//toString()                this.engg=engg;
---                       }
}
```

```
@Named("vehicle")
public class Vehicle{
@Resource(name="eng1") // performs byName mode autowiring
private Engine engg;   public void setEngg(Engine engg){
// toString()                this.engg=engg;
---                       }
}
```

Difference between @Autowired, @Inject and @Resource

@Autowired	@Inject	@Resource
⇒ Spring Annotation	⇒ JEE java config annotation	⇒ jdk java config annotation
⇒ Support byType, byName, constructor mode of Autowiring	⇒ Supports byName, byType, constructor mode of autowiring	⇒ Support byType, byName mode of autowiring
⇒ makes bean class as invasive class	⇒ Non-invasive class	⇒ Non-invasive class
⇒ Param required is available	⇒ No param	⇒ Param name is available
⇒ Use @Qualifier (or) @Named to resolve ambituity problem	⇒ Use @Qualifier (or) @Named to resolve ambituity problem	⇒ We can use name attribute of <resource> tag.

@Scope

⇒ This is spring annotation which can be applied at class level to specify spring bean scope.

```
@Named ("vehicle")
@Scope("singleton")

public class vehicle {
  ---
  ---
}
```

⇒ If the annotation param name is "value" then we can place the content of that param with out specifying param name.

@Named (value="vehicle") is equal to @Named("vehicle")

Note As of now spring is providing functionality only to few java config annotations.

20/7/15

applicationContext.xml

```
<context:component-scan base-package="com.nt.beans" />
```

@PostConstructor, @PreDestroy

⇒ Bean life cycle cfgs can be done in 3 ways

a) Declarative approach

(By using init-method, destroy-method attributes of <bean> tag)

b) Programmatic approach

(By implementing InitializingBean, DisposibleBean Interface)

c) Annotation approach

(By using java config annotations @PostConstruct, @PreDestroy)

@PostConstruct : To configure normal method custom init method

@PreDestroy : To configure normal method as custom destroy method

NOTE : Both methods should have the following signature

```
public void <method>() {
}
```

IOC Phase2 (@Required @PostConstruct, @PreDestroy)

Voter.java

```
@Named ("vt")
@Scope ("singleton")
public class Voter{
    @Value("raja")
   private String name;
    @Value("30")
   private int age;

   @PostConstruct
  public void myInit(){
      S.O.P("Voter : myInit()");
   if(name==null || age<=0){
      throw new IllegalArgumentException("name,age must be set");
      }
  }
@PreDestroy
   public void myDestroy(){
     S.O.P("Voter : myDestroy()");
      name=null;
      age=0;
    }
public String checkVotingEligibility(){
      if(age<18)
     return name+" ur not eligible to vote";
    else
    return name+" ur eligible to vote";
  }

}
```

applicationContext.xml

```
<context:component-scan base-package="com.nt.beans" />
```

clientApp.java

```
public class ClientApp {
    public static void main(String[] args) {
        //create IOC container
ApplicationContext ctx = new FileSystemXmlApplicationContext ("src/applicationContext.xml");
        //get Bean
    Voter vt = ctx.getBean("vt", Voter.class);
     String status = vt.checkVotingEligibility();
          S.O.P("Result ::: "+status);
    ((FileSystemXmlApplicationContext) ctx).close();
    }
}
```

IOCProj33(Anno-PropertiesFile)

- src
  - com.nt.beans
    - DBOperationsDAO.java
  - com.nt.commons
    - DB.properties
  - com.nt.cfgs
    - applicationContext.xml
- test
  - ClientApp.java

DBOperationsDAO.java

```
@Named("dao")
Public class DBOperationsDAO{
    @Value("${jdbc.drivel}")
    private string drivel;
    @Value("${jdbc.url}")
    private string url;
    @Value("${jdbc.username}")
    private string username;
    @value("${jdbc.pwd}")
    private string password;
  @Override
  public string toString(){
    letun "DBOperationsDAO[drivel="+drivel+", url="+url+", username="+username+",
                                    password="+password+"]";
      }
}
```

DB.properties

```
jdbc.drivel = oracle.jdbc.driver.OracleDriver
jdbc.url    = jdbc:oracle:thin:@localhost:1521:xe
jdbc.username = scott
jdbc.pwd = tiger
```

applicationContext.xml

```
<context:component-scan base-package="com.nt.beans"/>
<context:property-placeholder location="src/com/nt/commons/DB.properties"/>
```

clientApp.java

```
public class ClientApp{
    public static void main(String[] args){
ApplicationContext ctx = new FileSystemXmlApplicationContext("/applicationContext.xml");
DBOperationsDAO dao = ctx.getBean("dao", DBOperationsDAO.class);
        SO.P(dao);
    }
}
```

Wolding with properties file in Annotation Environment

⇒ To detect the classes and to recognize them as spring beans through annotations we can use <context : component-scan ...> tag.

⇒ To detect specified properties file .ie, cfg through annotation we can use <context : property-placeholder> tag.

⇒ To inject the values of properties file to bean properties we need to specify the keys of properties file through place holder ${<key>}

NOTE : Wolbing with <context : property-placeholder...> tag is equal to wolding with property placeholder configure bean class of spring obj.

Developing spring core Module Application Context with out xml file

⇒ For this we need to use "AnnotationConfigApplicationContext" to create IOC container by specifying "configuration" class.

⇒ The java class that contains method to create & return bean class obj is called "configuration" class.

⇒ We use @Configuration to configure oidinary class as the configuration class & we use @Bean on the top of configuration class methods which produces objects (To configure those objects as bean objects)

```
@Configuration
public class BeanConfiguration
{
  @Bean
  public WishGenerator createWishGenerator(){
  return new WishGenerator();
  }
}
```

In ClientApp.java

```
//IOC container
ApplicationContext ctx = ------.
WishGenerator wish = (WishGenerator) ctx.getBean(WishGenerator.class);
```

Limitations with above approach

1. Kills non-invasive behaviour of spring bean
2. Kills loose coupling
3. Configuring predefined, third party supplied classes as spring beans is very complex.

NOTE : The class ie, specified with @Configuration is called Configuration class. In that class we add @Bean on the methods that produce Bean class object. These methods will be executed automatically. The moment IOC container is started.

IOCProj34

```
|--> src
     |-> com.nt.beans
          |-> BeanConfigurator.java, WishGenerator.java, Student.java
     |-> test
          |-> ClientApp.java
```

BeanConfigurator.java

```
@Configuration
public class BeanConfigurator {

        @Bean(name="st")
        @Lazy(value=true)
        @Scope("prototype")
    public Student createStudent(){
          return new Student();
        }
      @Bean
      @DependsOn("st")
    public Date createDate(){
          return new Date();
       }
      @Bean
    public WishGenerator createWishGenerator(){
          return new WishGenerator();
      }
}//class
```

Student.java

```
public class Student {
        public Student() { }
    @Override
    public String toString() {
        return "101 -> Raja";
    }
}
```

WishGenerator.java

```
public class WishGenerator {
        public WishGenerator() { }
    @Override
    public String toString() {
        return "Good Morning";
    }
}
```

ClientApp.java

```
ApplicationContext ctx = new AnnotationConfigApplicationContext(BeanConfigurator.class);
```

Comments

⇒All the methods of configuration class that are annotated with @Bean will execute automatically the moment IOC container is created by specifying configuration class.

If we want to delay certain method execution until ctx.getBean() is called then use @Lazy annotation.

⇒In order to create Bean class object only after creating another bean class object then we can go for @Dependson

```
@Configuration
public class BeanConfiguration {
    @Bean(name="wish")
    @Lazy(value=true)
    @Dependson("dt")
    public WishGenerator createWishGenerator(){
        return new WishGenerator();
    }
    @Bean(name="dt")
    @Scope("singleton")
    public Date createDate(){
        return new Date();
    }
}
```

⇒Bean Instantiation that takes place when IOC container is created is called "Eager Instantiation."

⇒Bean Instantiation that takes place when ctx.getBean() is invoked is called "Lazy Instantiation".

21/7/15

SPRING JDBC (or) SPRING DAO

Q Where should we use jdbc and where should we use hibernate?

⇒ While working with jdbc if we want to process huge amount of records then we need to deal with single jdbc ResultSet object.

⇒ While working with hibernate if we want to process huge amount of records then we need to take multiple objects representing multiple records this is a problem and it may crash the system while creating huge amount of objects.

⇒ use jdbc in offline Appln's that deals with huge amount of data when donot need database protability feature. (changing DB s/w with out modifying the code).

Eg. Storing managing census details, Political party membership details by performing data validation and data correction.

⇒ While developing online Appln's (Internet web applications) we need that deals with limited amount of personalized data use hibernate.

Eg. online shopping, mobile recharge, online payments etc..

⇒ While developing Intranet appln's that performs batch processing of huge amount of data then use JDBC..

Eg: Courier Appln's that takes orders and process them at a time.

Eg: Mobile Number Activation Appln's that activates set of numbers at a time.

22/7/15

⇒ Spring jdbc provides abstraction layer on plain jdbc and simplifies persistence logic development.

⇒ Instead of using plain jdbc in off line Appln's and intranet Appln's that deals with huge amount of data through batch processing it is recommanded to use spring JDBC

Plain jdbc code

⇒ Register jdbc driver
⇒ Establishing the connection } (common logics)
⇒ create statement object
⇒ Send and execute SQL Query
⇒ Gather Results and process Results } (Application Logic)
⇒ Exception handling } [common logics]
⇒ close jdbc objects

Here we need to write both common logics and application specific logics in the application so it raises boiler plate code problem.

Spring JDBC Appln

⇒ Cfg and get Jdbc Template class object
⇒ Send and execute SQL Query
⇒ Gather results and process results

Note Here we need to take care of only Application specific logics development.

Limitations with plain Jdbc

⇒ Boiler plate code problem
⇒ Exception handling is mandatory
⇒ Processing ResultSet to get result in desired type is complex process
⇒ No support for Proper Transaction management.

Spring Jdbc/DAO advantages

⇒ solves boiler plate code problem
⇒ Makes programmer to take care of only application specific logics as shown above.
⇒ Raises unchecked exceptions and (DataAccessException and its subclasses) so the exception handling is optional. It is internally uses Exception rethrowing concept to convert checked Exception (SQL Exception) in to unchecked Exception.
⇒ Gives various methods to get the records of ResultSet in desired types.
⇒ Takes care of opening connection and closing connection activities automatically.
⇒ Supports both Named and positional parameters in SQL Query
⇒ Allows to use callback interfaces to develop more customized persistence logic by using plain Jdbc code..

Action	Programmer	SpringJDBC
⇒Configuring DataSource	yes	—
⇒Creating/getting connection	—	yes
⇒Creating statement	—	yes
⇒Preparing SQL Query with parameters	yes	—
⇒Sending and executing SQL query in database fw	yes	—
⇒Iterating through ResultSet (If needed)	—	yes
⇒Exception handling	—	yes
⇒Transaction management (If needed)	—	yes
⇒ Closing Jdbc objects	—	yes

Spring jdbc api packages

org.sf.jdbc and its sub packages

+

→ org.sf.dao and its sub packages

→ This package contains only Exceptions.

Note: Spring JDBC/ORM modules internally uses multiple other technologies/frameworks to develop persistence logics but translate the exceptions of underlying technology/framework to spring's DataAccessException hierarchy.

There are 5 methodologies/techniques to develop persistance logics in spring JDBC

1) Using JdbcTemplate
2) Using NamedParameterJdbcTemplate
3) Using SimpleJdbcTemplate
4) Using SimpleJdbcInsert, SimpleJdbcCall
5) Mapping SQL operations as subclasses using SqlQuery/SqlUpdate class (JDO style).

23/11/15

⇒ While working with any technique of spring JDBC we need to use jdbc connection to interact with database s/w.

⇒ It is recommanded to use jdbc con pool based jdbc connection instead of direct jdbc connection.

⇒ Donot use spring's built in classes like DriverManagerDataSource, SingleConnectionDataSource and etc. to work with jdbc connection because these classes actually does not pool the connection objects more over creates jdbc con objects on demand.

Q) What is the jdbc con pool that should be used in our spring Appln?

A) ⇒ If our spring Appln is standalone Appln running outside the server then use third party jdbc con pool like C3P0 (or) Apache DBCP and etc..

⇒ If our spring Appln is developed deployable in SERVER then use server managed jdbc con pool.

1) Using JdbcTemplate

⇒ This is central class of spring jdbc. provides abstraction layer on plain jdbc and simplifies persistence logic development.

⇒ This class takes care of common work flow activities of jdbc programming like opening connection, creating statement, closing connection and etc... but makes the programmer to just provide SQL query and to extract results.

⇒ This class translate jdbc generated SQLException into spring specific unchecked exception like DataAccessException. This allows programmer to avoid common errors in Application development like forgetting to close jdbc objects.

⇒ To create JdbcTemplate class object we need JdbcDataSource object.

⇒ We generally configure JdbcTemplate in spring bean cfg file and we inject that object to DAO class.

persistence-beans.xml

```
<bean id="drds" class="org.sf.jdbc.datasource.DriverManagerDataSource">
---
</bean>
<bean id="template" class="org.sf.jdbc.support.JdbcTemplate">
  <property name="dataSource" ref="drds" />
</bean>
```

⇒ There are two approaches of working with JdbcTemplate

a) Using direct methods (No need of plain jdbc code)

b) Using Callback interfaces (Need to write plain jdbc code to place customized persistence logic).

a) Using direct methods

⇒ Working with direct methods of JdbcTemplate class.

1. Select Operations (query, queryForXxx(-) methods)

a) queryForInt(), queryForLong()

To execute the query that gives single numeric value as the result.

Eg: select count(*) student.

b) queryForMap()

To execute the query that gives single record.

Eg: select * from student where sno = 1001.

c) queryForObject()

To execute the query and to get result into Pre-defined or BO class object.

Eg: select * from student where sno = 1001

d) queryForList()

To execute the query that gives multiple records

Eg: select * from student.

e) queryForRowSet()

To execute query that gives Jdbc RowSet object.

Eg:

and etc..

Note: If above methods are called with 1 argument then it internally uses jdbc Simple statement object

Note: If the above methods are called with 2 or 3 args then it internally uses jdbc PreparedStatement object.

2) Non-select operations

a) update(-)

3) batch updation

a) batchUpdate(-,-)

and many more....

DAO Prog 1 (JdbcTemplate-Directmethods)

```
|-> src
    |-> com.nt.service
        |-> DBOperations.java
        |-> DBOperationsService.java
    |-> com.nt.dao
        |-> DBOperationsDAO.java
    |-> com.nt.cfgs
        |-> persistence-beans.xml
    |-> test
        |-> ClientApp.java
```

jar files: IOCLib jar files + ojdbc14/6.jar
+ org.sf.jdbc-<version>.jar
+ org.sf.tx-<version>.jar
C3P0-0.9.1.2 + commons-pool.jar + common-dbcp.jar (spring 2.x)

[To work with C3P0 pool we require c3p0-0.9.1.2 jar file]

(Refer appno:31 & Page No:85)

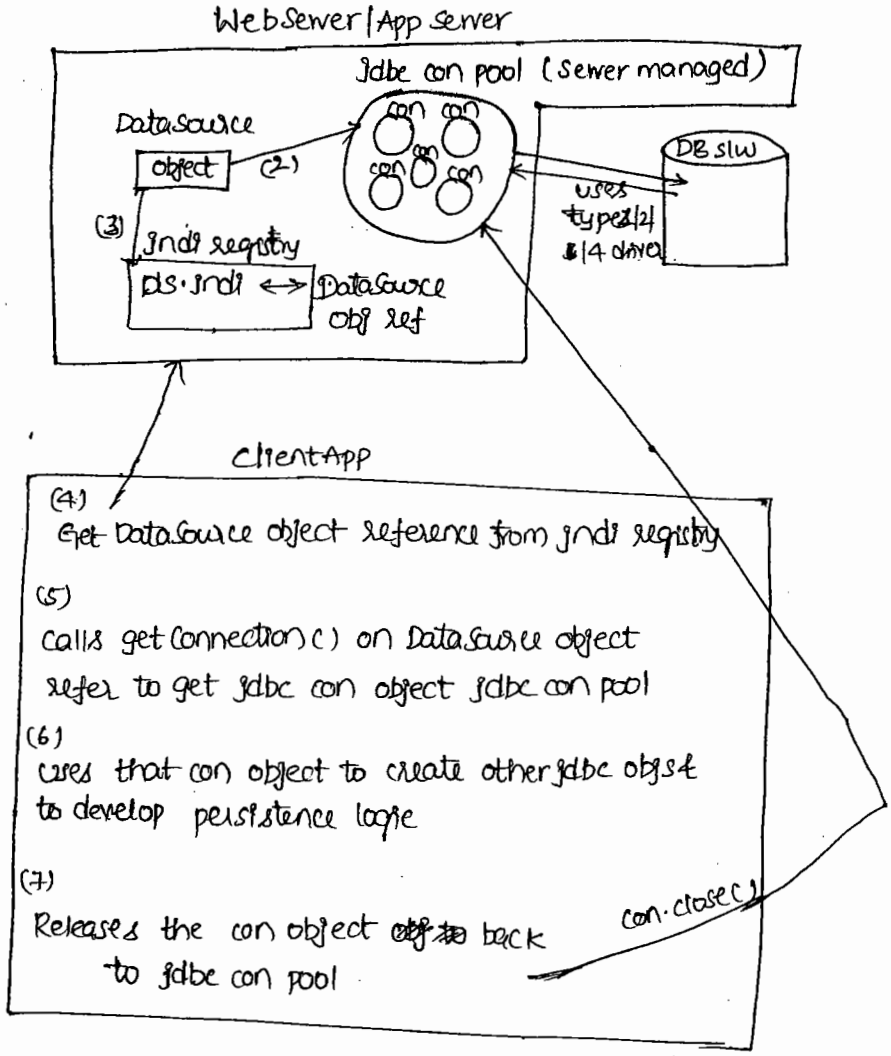
25/7/15 Connection pooling

⇒ To provide Global visibility to object or object reference it will be placed in jndi registry having nickname | jndi name.

⇒ Every webserver | application server gives one built-in jndi registry.

⇒ In server managed jdbc con pool environment, the jdbc con pool will be represented by DataSource object and its reference will be placed in jndi registry of server for global visibility having jndi name.

⇒ The Appln deployed in server or running from outside server can access DataSource object ref from jndi registry to get jdbc con objects from jdbc con pool.



Note In one server we can create multiple jdbc con pools belonging to same or different database softwares. All connection objects in con pool represents connectivity same database slw.

⇒ The client App that wants to use connection pool can be there outside the server as standalone appln and can be there inside the server as deployable servlet component, jsp component, ejb component.

Note W.R.to diagram

1) TL(or) PL(on Admin) makes server to use type 1|2|4 jdbc driver to interact with database s/w and to create server managed jdbc connection pool having connection objects

2) &(3) TL|PL|Admin creates datasource object representing jdbc connection pool and places that datasource object reference in jndi registry having jndi name for global visibility.

(4) to (7) refer the diagram.

⇒ In Weblogic s/w we can create multiple domains and each domain acts as 1 domain server

⇒ Multiple projects of a company uses multiple domains servers of weblogic s/w on 1 per project basis.. ie, the weblogic s/w will be instantiated only for 1 time. in a common computer and in that s/w multiple domains will be created for multiple projects on 1 per project basis.

Procedure to create user-defined Domain server in weblogic 12c. (compatable with jdk1.7)

⇒ Start → Oracle ~~Weblogic~~ → OracleHome → Weblogic server 12c → Tools → Configuration Wizard.

⊙ create new domain → D:|Oracle Weblogic 12c| user-projects |domains| Sp27Domain → Next → Next →

User Name [javaboss]

Password [javaboss1]

Confirm Password [javaboss1]

[Next] → [Next] →

☑ Administration server

[Next]

Listen Port [7080]

Next → Next → create (finish).

Procedure to create jdbc datasource pointing jdbc con pool for oracle in "Sp27Domain" server of weblogic

1) Start Sp27Domain Server

D:|Oracle Weblogic 12c| user-projects| domains | Sp27Domain |startWeblogic.cmd

2) Open Admin console of that domain server

http://localhost:7080/console →

username : javaboss

password : javaboss1

→ login

(3) create jdbc datasource pointing to jdbc con pool for oracle.

Admin Console screen → services → Data Sources → New → Generic DataSource →

Name : ds1

JNDI Name : DsJndi

Database Type : oracle

Next

Database Driver : Oracle's Driver (Thin) for service connections

Next → Next

Database Name : xe

Host Name : localhost

Port Number : 1521

Database User Name : scott

Database Password : tiger

Confirm Password : tiger

→ Next

Test Configuration → Next

☑ Admin server

Finish

Connection Pool

Initial capacity : 3

Maximum capacity: 30

Minimum capacity: 0

Shrink Frequency: 900

save

Shrink Frequency: After every 900 seconds, the idle con objects are destroyed automatically.

⇒ Spring gives 1 built-in factory Bean class to get object from jndi registry based on the given jndi name. This factory bean class name org.sf.jndi.JndiObjectFactoryBean.

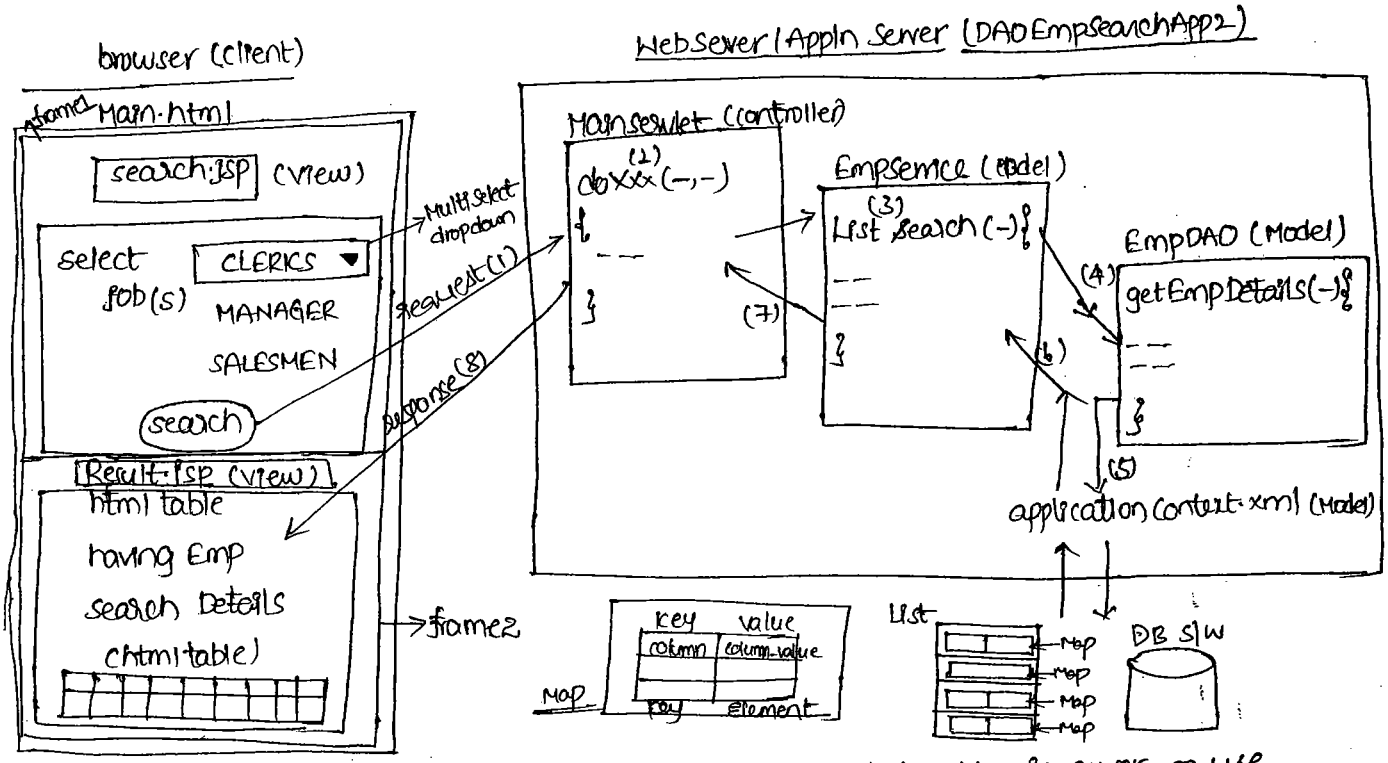
eg In Bean cfg file

```
<bean id="jofb" class="org.sf.jndi.JndiObjectFactoryBean">
    <property name="jndiName" value="DsJndi"/>
</bean>
```

gives dataSource object gathered from jndi registry.

```
<bean id="template" class="org.sf.jdbc.core.JdbcTemplate">
    <property name="dataSource" ref="jdbc"/>
</bean>
```

↳ since the above bean is factory bean. The gathered dataSource object will be injected.



⇒ The above application is web application so it is deployable in server so use server managed jdbc con pool

DAOEmpSearchApp2 (Eclipse Dynamic Web Project)

```
|--> Java Resources
    |--> src
        |--> com.nt.service
            |--> EmpService.java
        |--> com.nt.dao
            |--> EmpDAO.java
        |--> com.nt.cfgs
            |--> applicationContext.xml
        |--> com.nt.controller
            |--> MainServlet.java
|--> WebContent
    |--> Main.html, Search.jsp, Result.jsp
    |--> WEB-INF --> web.xml
```

jar files : SPJDBC LIB jar files + ojdbc14.jar + jstl.jar + standard.jar
(Perform Deployment Assembly). → collect from Tomcat installation.

(Refer Appln 33 for page 91)

⇒ While developing Eclipse based web application, the jar files placed in buildpath will not be moved to WEB-INF/lib folder automatically. For that we need to perform Deployment Assembly separately.

Right on the project → build path → configure build path → Deployment assembly → add → Java Build Path entries → select jar files → finish → Apply → ok.

⇒ To deploy Eclipse webproject in Ntsp27 Domain of weblogic

1) Start Ntsp27 domain server

<weblogic_home>\user-projects\domains\Ntsp27Domain\startweblogic.cmd

2) Export web file Ntsp27Domain server

Right click on project → Export → war files →

webproject : CurrentProject name

Destination : <weblogic_home>\user_projects\domains\Ntsp27Domain\autodeploy\DAOEmpSearchApp2.war

finish).

3) Test the Application

http://localhost:7080/DAOEmpsearchApp2/Main.html

JSP Page (result.jsp)

```
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
 <c:choose>
  <c:when test="${not empty result}">
    ↳ checks whether list collection is empty or not
   <table>
    <tr> <td> NO </td><td> Name </td> <td> JOB </td> <td> SALARY </td> </tr>
   <c:forEach var="map" items="${result}">
    ↳ points to one element of list collection in each iteration.
    ↳ request attribute holding list collection
    <tr>
      <td> <c:out value="${map.EMPNO}"/>
      <td> <c:out value="${map.ENAME}"/>
      <td> <c:out value="${map.JOB}"/>
      <td> <c:out value="${map.SAL}"/>
    </tr>
   </c:forEach>
  </table>
</c:when> <c:otherwise> Employee Not Found </c:otherwise> </c:choose>
```

Procedure to create userdefined Domain in Glassfish 4.x

admin username

<Glassfish4.x_home>\bin> asadmin create-domain --adminport=4646 --user=testuser

admin console port

GFNtsp27Domain

Domain Name

enter password : testuser

Re-enter Password : testuser

--

--

Domain created successfully.

⇒ To install Glassfish 4.x Extract the zip file.

⇒ Glassfish uses separate port no for http operations and seperate port no for admin operations

⇒ Procedure to create jdbc datasource pointing to jdbc con pool for oracle in GFNtsp27Domain server of Glassfish

1) Place ojdbc14.jar file in <Glassfish_home>\domains\GFNtsp27Domain\lib\ext folder.

2) start the domain server

<Glassfish_home>\bin> asadmin start-domain GFNtsp27Domain ↵

command executed domain server started successfully.

3) Open admin console of domain server

http://localhost:4646

username : testuser

password : testuser

(login)

4) Create jdbc con pool for oracle

Admin Console → Resources → JDBC Connection Pools → New →

Pool Name : [pool1]

Resource Type : [javax.sql.DataSource]

Vendor : [Oracle]

[Next]

Initial & Minimum Pool size : [8]

Maximum Pool size : [100]

Max Wait Time : [60000] Millisecond

☑ User : scott

☑ DatabaseName : xe

☑ password : tiger

☑ serverName : localhost
☑ URL : jdbc:oracle:thin@localhost:1521:xe
☑ portName : 1521

Finish

Lauch pool1, Ping ✓

5) Create jdbc Datasource pointing to the above jdbc con pool

Admin console → Resources → JDBC Resources → New →

JNDI Name: DSJndi
Pool Name : Pool1

ok ✓

Test the application

⇒ To deploy application GFNtsp27 Domain server of glassfish place directly our war file in <GlassFish_Home>\Domains\GFNtsp27Domain\autodeploy folder.

⇒ http://localhost:6565/DAOEmpsearchApp2/Main.html

Select Job(s): MANAGER ▼ search
CLERK
SALESMAN

O/P

NO	NAME	JOB	salary
101	Raja	MANAGER	10000

30/7/15

Creating jdbc connection pool for oracle in Tomcat 8 server.

Step-1) Make sure that tomcat-jdbc.jar file is available in <Tomcat_home>\lib folder.
NOTE This jar file is built-in file in Tomcat 8. But we need to arrange separately for other versions of Tomcat by extracting apache-tomcat-jdbc-1.1.0.1-bin.zip file.

Step-2) place ojdbc14/6.jar file in <Tomcat_home>\lib folder.

Step-3) Place the following <Resource> tag in <Tomcat_home>\conf\context.xml file as the sub tag of <context> tag.

<Resource name="mypool" (jndiname) type="javax.sql.DataSource" factory="org.apache.tomcat.jdbc.pool.DataSourceFactory" driverClassName="oracle.jdbc.driver.OracleDriver" url="jdbc:oracle:thin:@localhost:1521:xe" username="scott" password="tiger" initialSize="10" maxActive="20" maxIdle="10" minIdle="5" maxWait="100" validationQuery="SELECT SYSDATE FROM DUAL">

Step-4) Restart the server.

NOTE: To use the above jdbc connection Pool with tomcat server we need to place

Note To use the above jdbc con pool in web application we need to place jndi name as below,

java:/comp/env/mypool
(fixed prefix) (jndi Name)

Procedure to configure Tomcat 8 server with eclipse IDE.

Window → Preferences → Server → RunTimeEnvironments → Add → Apache Tomcat v8.0 → Next → Browse → D:/Tomcat 8.0 (Tomcat installation directory) → ok.

Note If we deploy any webappln in Tomcat server through eclipse IDE by configuring Tomcat server with eclipse IDE it is recomanded to place context.xml file in servers folder of package Explorer in order to use the Tomcat Server Managed jdbc con pool in that web appln.

Callback Interfaces

⇒ The method that is called by underlying JVM/container/server/etc... automatically and dynamically is called callback method. The interface that contains such methods is called callback interface.

⇒ Spring provides lots of callback interfaces along with lots of Template classes. We can use these callback interfaces to write customised logics to process data and to get data in customized manner.

For ex the query forMap(-) is called single record in the form of Map object having column names as the keys and col values as the values in the map datastructure elements. But industry standard is getting that record as BO class object. For this spring gives RowMapper callback interfaces...

⇒ Various callback interfaces of spring jdbc are

- StatementCallback → Allows to work with statement object
- PreparedStatementCallback →
- CallableStatementCallback →
- PreparedStatementCreator → Provides con object to create PreparedStatement
- PreparedStatementSetter → provides PreparedStatement obj to set values to query params
- ✓ RowMapper → To process 1 record of ResultSet at a time
- ✓ ResultSetExtractor → To process multiple records of ResultSet at a time..

Note The implementation class methods of callback interfaces will be called by JdbcTemplate internally...

⇒Example Appln to get single Employee record as EmployeeBO class object and multiple Employee records as multiple EmployeeBO class objects stored in List Collection.

DAOProject3 (CallbackInterfaces)

```
|--> src
    |--> com.nt.cfgs
        |--> applicationContext.xml
    |--> com.nt.service
        |--> ListEmpService.java
    |--> com.nt.BO
        |--> EmployeeBO.java
    |--> com.nt.dto
        |--> EmployeeDTO.java
    |--> com.nt.dao
        |--> ListEmpDAOImpl.java (class),
        |--> ListEmpDAO.java (Interface).
    |--> test
        |--> clientApp.java
```

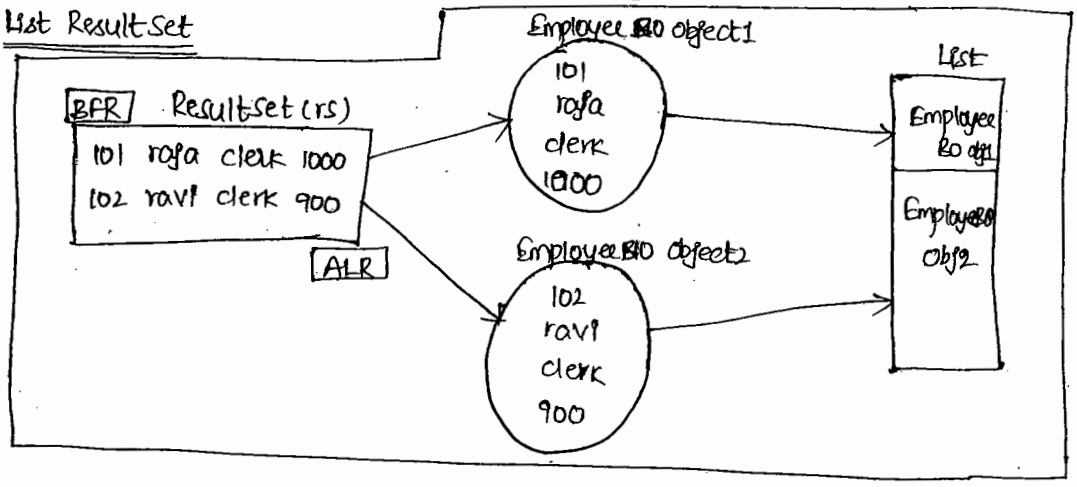
jar files : SPJDBC LIB jar files +

31/7/15

⇒ Two methods of JdbcTemplate to work with RowMapper, ResultSetExtractor Callback interfaces for the above requirement.

1) public Object queryForObject (String sql, Object args[], RowMapper<T> rowmapper)
   → useful to get BO class object having single record

2) Public List<T> query (String sql, Object args[], ResultSetExtractor<T> extractor)
   → useful to get BO class objects having multiple records.



⇒ In realtime, the end user supplied data will be received by service class in the form of VO (or) DTO class object data. After processing request BO class object will be prepared to pass to DAO. DAO uses BO class object data for persistence operation and gives another BO class object back to service class. This service class converts BO class object to DTO class object data to send to client.

⇒ VO, BO class objects are not serializable to send over the network whereas DTO class objects are serializable objects.

9/08/15

Batch processing / updation

⇒ Instead of performing multiple insert, update, delete operations on DB table by sending same query to database slw for multiple times it is recomanded to take the support of batch processing. which sends query to database slw for one time and sets batch of values to query for one time to reduce network round trips blw java appln and db slw.

⇒ While working with jdbc template we use batchUpdate(-) method for this.

1) batchUpdate(String sql, List<Object[]> args)

2) batchUpdate(String sql, PreparedStatementBatchSetter psbs)

3) batchUpdate(String sql, List<Object[]> args, int[] args types) and etc...

⇒ PreparedStatementBatchSetter is callback interface that can be used to set values on a PreparedStatement provided by JDBC Template for each of a number of ~~non-select queries~~ updates in a batch using same SQL. It allows the programmer to set the batch values of query parameters in customized form by gathering values from JAVA objects like BO objects not from object array. This call back interface gives 2 methods.

a) setValues(PreparedStatement ps, int index)
   ↦ Allows to set values to query params by specifying its index and by specifying Query index to execute

b) getBatchSize()
   ↦ returns numeric value to specify the no. of times that setValues(-,-) should execute

Example If getBatchSize() returns 5 then setValues(-) method executes for 5 times to assign 5 sets of values to batch, later the queries executes multiple times in DB slw having those of values.

ClientApp

service.registerStudent(List studdto) ⟶

Service class

public String registerStudents(List studdto){

---

convert ListStuddto to List studbo object

dao.insert(List studbo);

}

↓

DAO class

public boolean insert(List studbo){

--

}

↓

DAOProj4 (Batch Updation)

- src
  - com.nt.cfgs
    - applicationContext.xml
  - com.nt.bo
    - StudentBO.java
  - com.nt.dto
    - StudentDTO.java
  - com.nt.service
    - RegisterStudentsService.java
  - com.nt.dao
    - RegisterStudentsDAO.java
  - test
    - ClientApp.java

jar files : Spring JDBC Hb

DB Table: Student

- sno(n) (pk)
- sname(vc2)
- sadd(vc2)

⇒ In order to use local variables (or) parameter method definition inside the innerclass we must take them as final variables.

02/08/15

2) NamedParameter JdbcTemplate

1. Spring Jdbc supports 2 types of params in the SQL query
   a) PositionalParameter(?)
   b) NamedParameter(:name)

2. JdbcTemplate supports only positional parameters(?) we need to use var args values (or Object[] values to assign values of positional parameters based on their indexes. In this process if we confuse towards the indexes then there is possibility of getting problems.

3. To overcome above problem the spring jdbc has given NamedParameterJdbcTemplate as wrapper around JdbcTemplate supporting NamedParameter(:varname)

   Ex: select * from emp where empno>=:min and empno<=:max;

4. The underlying Jdbc of used by spring Jdbc converts each name parameter into positional parameter before executing the query.

5. There are 2 ways to set values of NamedParameter
   a) As Map<String, Object> obj
      Specify the names of NamedParameters as keys and the objs as values in elements

      Ex:
```
Map<String, Object> map = new HashMap<String, Object>();
map.put("min", 10);
map.put("max", 100);
```

   b) By using the implementation class object of SqlParameterSource (Interface)

      i) MapSqlParameterSource

      ⇒ Allows the specify the name, values of NamedParameter as the ~~range~~ args of ~~addValue~~ addValue(-,-).

```
SqlParameterSource params = new MapSqlParameterSource();
params.addValue("min",10);
params.addValue("max", 100);
```

      ii) BeanPropertySqlParameterSource

      ⇒ Allows to specify Bean object values in NamedParameter values but the names of NamedParameters and BeanProperty Names (Member variables) must match.

DAOProj5 (NamedPositionalParameter)

```
|--> src
    |--> com.nt.service
        |--> EmployeeService.java
    |--> com.nt.dao
        |--> EmployeeDAO.java
    |--> com.nt.dto
        |--> EmployeeDTO.java
    |--> com.nt.bo
        |--> EmployeeBO.java
    |--> com.nt.cfgs
        |--> applicationContext.xml
|--> test
    |--> ClientApp.java
```

03/08/15

3. SimpleJdbcTemplate

⇒ Introduced in spring 2.5, supporting as alternate to JdbcTemplate. It supports all jdk1.5 features like autoboxing, autounboxing, var args etc....

⇒ In spring 2.5 version JdbcTemplate did not supported to jdk1.5 features. spring 3.x onwards JdbcTemplate class supports to jdk1.5 features, so SimpleJdbc Template class is deprecated in spring 3.x & 4.x.

Working with SimpleJdbcInsert, SimpleJdbcCall

⇒ use metadata concepts as simplifies environment to call plsql procedures and to insert records

a) SimpleJdbcInsert

⇒ It uses DB tables metadata concepts to insert records dynamically without insert sql query it just takes tablename, column names, column values from programmer and use them to generate dynamic SQL insert query to insert record. We can give these column names and values either as map object (or) as sql parameter source implementation class object. The map object contains column names as the key and column values as values.

⇒ To create SimpleJdbcInsert object DataSource object is Dependent object and we can setTable(-) to specify db table name and execute(-) to specify column names, column values and also to execute the dynamically generated insert sql query.

⇒ If an appln deals with only registrations (insert operations) then we need to work with SimpleJdbcInsert class instead of JdbcTemplate class.

DAOProg 6 (SimpleJdbcInsert)

```
|--> src
    |--> com.nt.cfgs
        |--> applicationContext.xml
    |--> com.nt.service
        |--> StudentService.java
    |--> com.nt.dao
        |--> StudentDAO.java
    |--> com.nt.bo
        |--> StudentBO.java
    |--> com.nt.vo
        |--> StudentVO.java
    |--> test
        |--> clientApp.java
```

⇒ VO class is a java bean that does not implements Serializable (Interface) and its object represents input values coming to the Application.

⇒ DTO class is a java bean that implements Serializable (Interface) and its object represents i/p values (or) o/p values to from 1 resource to another resource over the network.

⇒ Generally the VO class object string properties working all the i/p values given by end-user like form data.

⇒ VO class generally contains string properties, DTO class contains all types of properties and BO class contains all types of properties (But these properties and DB table column type generally should match)

⇒ Simple Jdbc insert internally uses JdbcTemplate while completing the persistance logic.

⇒ JdbcTemplate, SimpleJdbcTemplate, NamedParameterJdbcTemplate class objects are thread safe once configure that means they allow one thread at a time to perform persistence operation which degrade to performance.

⇒ To overcome this problem we can use SimpleJdbcInsert (or) SimpleJdbcCall whose objects are multithreaded allowing multiple threads simultaneously.

Use case When we ask visitors to register in a time frame (like online web-caunding) we need to allow simultaneously record insertion. for better performance so prepare using SimpleJdbcInsert in that situation.

04/08/15

SimpleJdbcCall

⇒ It is a multithreaded, reusable object representing call to PLISQL procedures or function. It returns or runs on metadata concepts and simplifies the code that is required to call procedure or function. We just needed to provide procedure / fuction name with map of in parameters to call procedure or function and this returns a map having out parameter names & values.

⇒ Instead of writting common persistance logic in every module as SQL queries, it is recomanded to write only one time in database slw as PLISQL procedure or function.

Ex: Instead of writting Authentication module in every module seperately. write it only once as PLISQL procedures or function and used in multiple modules of a project.

Sample code

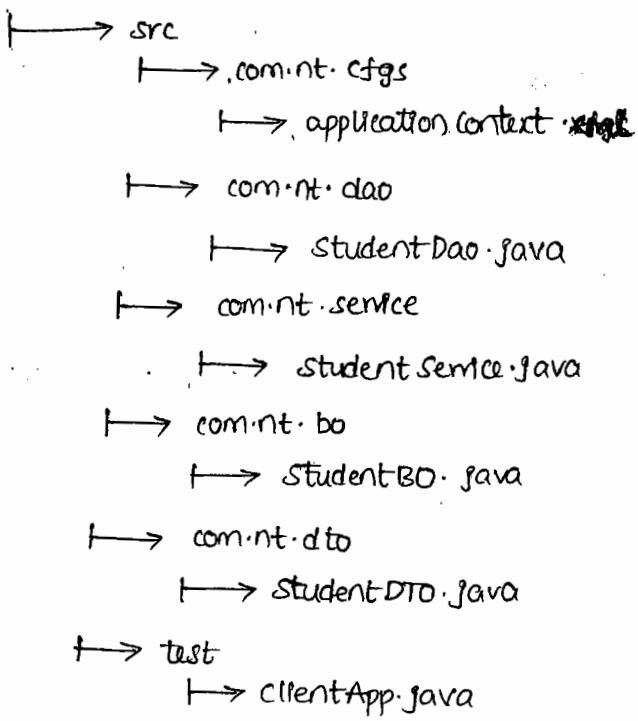
PLsql procedure in oracle

```
create or replace procedure getStudentDetails
        (no in number, name out varchar, address out varchar) as
 begin
 select sname, sadd into name, address from student where sno = no;
 end;
```

In DAO class

```
Private SimpleJdbcCall sjc;
 public void setSjc(SimpleJdbcCall sjc)
        {  this.sjc = sjc;
        }
public StudentBO getStudentDetails (int id){
        sjc.setProcedureName("getStudentDetails");
    // prepare IN params
 Map<String, Object> inParams = new HashMap<String, Object>();
                inParams.put("no", id);
          // call PLISQL procedure
 Map<String, Object> outParams = sjc.execute(inParams);
        // store the outparam values in BO object
 StudentBO bo = new StudentBO (id, (String)outParams.get("NAME"), (String)outParams.get("ADDRESS"));
   return bo; } // method
```

DAO Proj 7 (SimpleJdbcCall)



Assignment : use SimpleJdbcCall appoit to call PLSQL procedure/function that perfoums authentication.

5) Mapping SQL operations as subclasses

⇒ This is last techniques in spring jdbc to perform persistence logics. This technique make programmer to develop sub classes where we set DataSource, declare parameters and compilation of query only for 1 time but we execute Query for multiple times by re using the object of subclass for multiple times. This concept is similar to JDO programming.

⇒ Spring JDBC supplies SQLQuery, SqlUpdate classes to develop these sub classes.
Use SqlQuery for select operations, Use SqlUpdate for non-select operations.

⇒ While working with JdbcTemplate, NamedParameterJdbcTemplate, SimpleJdbcTemplate classes if same query is executed for multiple times then it creates multiple jdbc PreparedStatements internally on 1 PrepareStatement object for each execution basis. This is bad pratice and also degrade the performance. To overcome this problem we can use "Mapping SQL operations as subclasses" Technique.

1) SqlQuery class

⇒ We develop separate sub class extending from SqlQuery for each select Query inside DAO class. This sub class supplies data source, query to superclass to execute query for multiple times by compiling the query for only 1 time. SqlQuery supplies execute(-) methods to execute

Query that gives list of objects (multiple records) and also give finder methods to execute query for gives only 1 object (one record). This class also give mapRow(-,-) method which can be used to process each record of ResultSet object. The execute, finder methods internally calls mapRow(-,-) method. Since SqlQuery is abstract class having male abstract methods we prefer using "MappingSqlQuery" which is a subclass in the inheritance hierarchy of SqlQuery.

⇒ While developing sub class for SqlQuery (MappingSqlQuery) classes when we use the following the methods in the sub classes.

a) declareParameter(-) b) compile() c) executeMethodXxx(-) d) findXxx(-) methods..

⇒ This technique gives every thing to super class from subclass for query execution and reuse subclass object.

DAO Proj8 (MappingSQLOperations)

```
|---> src
      |---> com.nt.cfgs
            |---> applicationContext.xml
      |---> com.nt.bo
            |---> StudentBO.java
      |---> com.nt.dto
            |---> StudentDTO.java
      |---> com.nt.dao
            |---> StudentDAO.java
      |---> com.nt.service
            |---> StudentService.java
      |---> test
            |---> clientApp.java
```

jar files : same

refer DaoProj8 of page no: 20. of booklet 2.

→ selector = new StudentSelector(ds); → must be created in Constructor in order to use subclass object for multiple times in DAO class methods

→ selector.findByAddress(addrs) → calls subclass method to execute the query.

→ StudentSelector() → 2 constructor → supplies details to super class to use prepared stmt internally for compilation of the query by sending to DB s/w.

→ mapRow() → executes for each record processing of result set.

→ super.execute() → This method takes the query to super class and sets given argument to

query parameter, executes the query and gets result set objects and calls mapRow() method for multiple times to process multiple records of result set to get multiple BO objects and places those BO objects in List collection.

SqlUpdate class

⇒ It is similar to SqlQuery class but given for non-select queries (update queries) execution. We can prepare sub class to SqlUpdate class inside the DAO class where we supply datasource, query to SqlUpdate class only once but we use this subclass object for multiple times to execute same non-select query for multiple times with the support of update(-,-) methods.

⇒ SqlUpdate is a concrete class and supplies lots of update(-) methods to execute non-select Query with same or diff parameters for multiple times.

⇒ For every non-select query we need to take separate subclass for SqlUpdate inside the DAO class...

Refer DAO proj8 of page no: 22.

updator = new studentUpdate(ds, UPATE_STUDENT_ADDRS_BY_NO) → sub class object

int cnt = updator.updateDetails(newAddrs, no)

- studentUpdator(DataSource ds, String qry) → constructor
  - sends datasource, qry to super class and compiles the query in super class.

- int cnt = super.update(newAddrs, no)
  - takes already supplied non-select qry and executes that query having given arguments as the parameter values.

Q Why should we learn SimpleJdbcInsert, SimpleJdbcCall and MappingSqlOperations as subclasses technique eventhough JdbcTemplate class is already available?

A JdbcTemplate class object is singlethreaded object/thread safe object that means it allows one thread at a time to perform persistence operation. To overcome this problem we use SimpleJdbcInsert, SimpleJdbcCall classes.

⇒ When we use JdbcTemplate class obj to execute same query for multiple times. It internally creates multiple PreparedStatement obj on one object per each execution basis which is a bad practise to overcome this problem use MappingSqlOperations as subclasses technique

AOP (Aspect Oriented Programming)

⇒ AOP is based on decorator, proxy design pattern.

⇒ AOP is the methodology of programming that makes the programmer to separate primary logics of appln from secondary logics | helper logics.

⇒ AOP is not replacement for OOP, More over it complements OOP.

⇒ In OOP we develop business methods in classes by mixing both primary, secondary logics due to this business methods becomes heavy and the reusability of secondary logics will be killed and we can not enable or disable secondary logics without source code of Business methods.

⇒ To overcome the above problem use AOP that makes the programmer to write primary logics in separate class and secondary logics in separate classes and allows them to mixup | bind both logics dynamically at runtime by generating new classes.

OOP example

```
class Bank{

public boolean withdraw (int accno, int amt){
    Security Logic
    Logging Logic          } → secondary logic | Helper Logic
    Transaction Logic
    bal = bal - amt;       ⟼ primary logic
}
public boolean deposit( int accno, int amt){
    Security Logic
    Logging Logic          } → secondary logic
    Transaction Logic
    bal = bal + amt;       ⟼ primary logic
}
}
```

AOP Example

```
class Bank{
public boolean withdraw(int accno, int amt){
    bal = bal - amt;
}
```

public boolean deposit(int accno , int amt){

bal=bal+amt;

}

}

class Security {

Security Logic

}

class Logging {

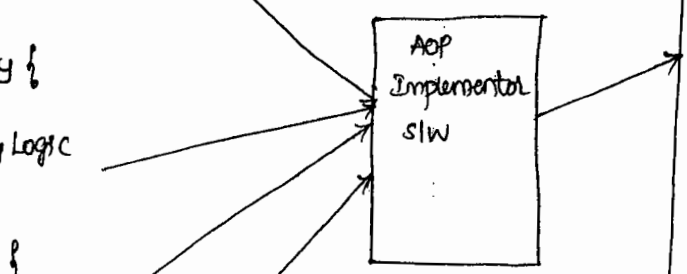
Logging logic

}

class Transaction{

Transaction Logic

}



Proxy class

class Bank $ Proxy {

withdraw (accno,amt){

Security Logic ;

Logging logic ;

Transaction Logic ;

bal=bal-amt;

}

deposit (accno,amt){

security Logic ;

Logging Logic ;

Transaction Logic ;

bal=bal+amt;

}

}

Notes

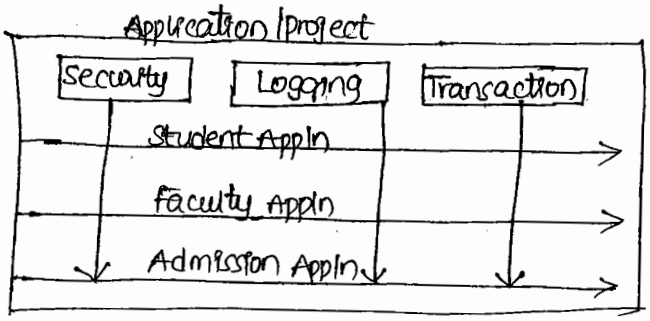
⇒ The logic that is minimum to complete the task is called primary logic. Without primary logic task will not be completed.

⇒ The Logic that supports primary logic and makes primary logic by better logic but optional to have in the application is called secondary logic.

Ex ⇒ While performing withdraw operations deducting amount from balance is called "primary logic".

Ex ⇒ checking the identity of user while logging and etc are called "secondary logic"

⇒ since we need secondary logics in multiple component classes of appln / projects these can be also called as cross-cutting concerns. Because they need to be applied across the multiple classes.



⇒ For example scenario that speaks about need of AOP with bank account usecase refer page no: 1,2,3,4 of AOP in material-2.

*AOP is not replacement for OOP, More ever it compliments oop.

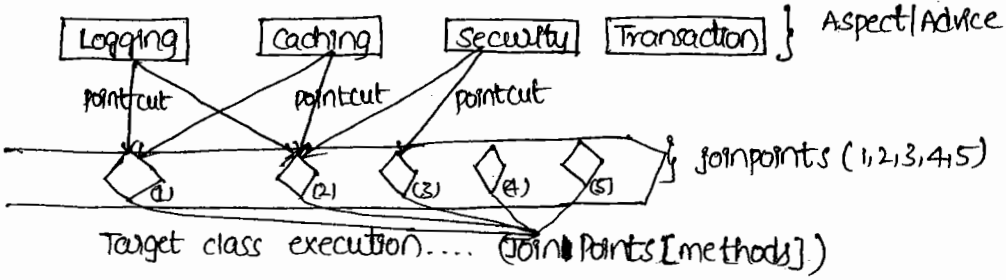
⇒ The key component in oop is class, the key component in AOP is Aspect (Cross cutting concern that represents primary logic).

⇒ The language that is designed based on OOP principles like abstraction, encapsulation, inheritance and etc.. is called <u>OOP enabled languages</u>

⇒ Similarly to OOP principles we also got AOP principles. The sw that supports and implements AOP principles is called <u>AOP enabled sw</u>. AOP is not just specific to spring. There are multiple AOP enabled sw like Spring AOP, AspectJAOP, Jboss AOP, JAC (Java aspect Components), Hugzilla AOP and etc..

<u>AOP Principles / Terminologies</u>

a) Aspects
b) JoinPoint
c) Advice
d) Pointcut
e) Target class
f) Weaving
g) Proxy class



a) <u>Aspect</u>: The logic that has to be applied across the multiple class of an appln is called <u>Aspect</u>. These logic is called as <u>Crosscutting concern, secondary logic, middleware services</u>
Eg: security, logging etc ..

b) <u>JoinPoint</u>: The possible points in classes on which aspect logics can be applied is called <u>JoinPoint</u>. They are like fields, constructors, methods and etc..

<u>Note</u>: spring supports only <u>methods as JoinPoint</u>.

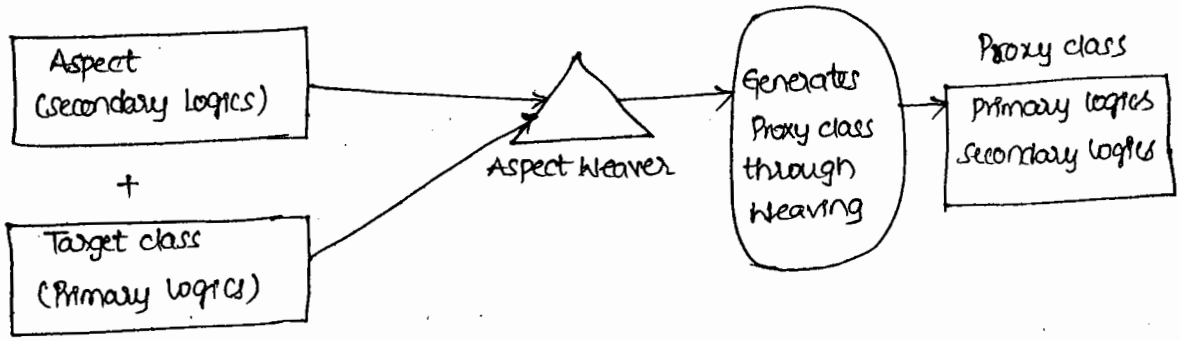
c) <u>Advice</u>: It specifies / indicates how an aspect should be applied on Join point. like applying before executing method or applying at the end of method and etc.. There are 4 types of advices like BeforeAdvice, AfterAdvice, ThrowsAdvice, AroundAdvice.

d) <u>Pointcut</u>: It is a collection of join point on which aspects are adviced (i.e., aspects are configured)

e) <u>Target class</u>: The class on which we want to advice the aspects. It is a pre-AOP java class having primary logics in methods. If we call call methods of target class only primary logics will be executed.

8) Weaving: The process of advicing aspects on the target class to build a proxy called class is called Weaving. It is all about combining multiple things to build 1 proxy class.

9) Proxy class : The outcome class of weaving process where primary logics and secondary logics will be mixed up dynamically at run time. It is post-AOP class. If we call methods on proxy class object the primary logics will be executed along with secondary logics.



⇒ In spring joinpoints are methods of classes where aspects can be advised. Pointcut specifies which aspect/aspects should be applied on which joinpoint/joinpoints.

08/08/15

⇒ Aspect Weaver is an unit of AOP framework software like spring AOP, AspectJ AOP and etc.. It uses some runtime libraries like JDK libraries (or) cglib libraries to generate proxy classes through weaving. These proxy classes are dynamically generated in memory classes.

⇒ Spring 1.x gives spring AOP to work with AOP. From spring 2.x AspectJ AOP has been integrated with spring so from spring 2.x, we can use spring AOP and AspectJ AOP to work with AOP concepts.

⇒There are 4 approaches to implement AOP concepts in spring environment.

a) Spring AOP Declarative (xml code)

b) Spring AOP Programatic (Java code)

c) Spring Integrated AspectJ AOP Declarative (xml)

d) Spring Integrated AspectJ AOP Annotation.

⇒There are 4 types of Advices in AOP style programming.

a) Around Advice: Here Advice logic executes around the target class method ie., before and after target class method (target method).

b) Before Advice: Here Advice logic executes before executing the target class method. so once target method is executed control does not come back Advice logic/ method.

c) After Advice: Here Advice logic executes after executing the target method and before it returns the control back to caller.

d) Throws Advice: Here the advice logic executes when exception is raised in target method.

a) Spring AOP Declarative Approach

⇒ Here we used xml files to configure Aspect, target classes and to configure the classes that perform weaving.

1) Around Advice

⇒ Advice indicates the action taken by Aspect on joinpoint so Around Advice makes its aspect to executes around the join point ie, Aspect/advice logic execution before entering to target method and after executing the target method.

⇒ This AroundAdvice is useful to implement "Performance Monitoring", "Caching", "Logging", and etc... use cases

⇒ To develop this advice/ aspect we need to take a class that implements MethodInterceptor() should provide invoke (MethodInvocation) as shown below.

```
public class MyAroundAdvice implements MethodInterceptor{
        public Object invoke(MethodInvocation invocation){
                --// logic that executes before entering into target method
            Object returnVal = invocation.proceed(); //calls target method
                --// logic that executes after finishing the execution of target method
        }
}
```

⇒ Using MethodInvocation object we can call getName(), getArguments() to target method name and its arguments. We can call proceed() method to call target method from advice class.

Important points | Control points of Around Advice

⇒ can access and modify target method arguments

⇒ can control target method execution ie, can skip (or) proceed target method execution being from advice class.

⇒ can access and modify the target method return value.

AOP Proj1

```
|---> src
      |---> com.nt.cfgs
            |---> applicationContext.xml
      |---> com.nt.aspect
            |---> LogAroundAdvice.java  (aspect | advice class)
      |---> com.nt.service
            |---> IntrAmtCalculator.java  (target class)
      |---> test
            |---> ClientApp.java
```

jar files : IOC jar files + spring-aop <version>.release.jar
+ spring-aspects <version>.release.jar
+ com.springsource.org.aopalliance-1.0.0.jar

Procedure to develop Spring AOP Application in Declarative Approach

a) Developing Target class having methods with primary logics

b) Developing Aspect | Advice class having secondary logics

c) Perform configuration in spring Bean configuration file for Weaving

(i) configure target class

(ii) configure Aspect | Advice class

(iii) configure ProxyFactoryBean class supplying target, Aspect classes to generate Proxy class through weaving.

d) Develop the client Appln for testing

(i) create IOC container

(ii) get the Proxy class object by supplying the bean id of "ProxyFactoryBean"

(iii) call target class methods on proxy class object

09/08/15

AOPProj1 (Internal flow)

Flow of execution with respect to AOPProj1 of material by involving the generated proxy class.

Target class

```
public class IntrAmtCalculator{
    public float calcIntrAmt(float p, float r, float t){ (g)
     return p*t*r/100.0f ; (h)
     }
}
```

Advice class

```
public class LogAroundAdvice implements MethodInterceptor {
        public Object invoke(MethodInvocation invocation){ (e)
          S.O.P("---");
      (i) Object retValue = invocation.proceed(); (f)
          S.O.P("___");
          return retVal ; (j)
        }
}
```

Proxy class

```
public class IntrAmtCalculator$Proxy extends IntrAmtCalculator
    {
   public float calcuIntrAmt(float principle, float rate, float time){ (c)
     MethodInvocation invocation = new MethodInvocation();
          invocation.setMethod("calcIntrAmt");
          invocation.setTargetClass("IntrAmtCalculator");
          Object args[] = new Object[3];
            arg[0] = principle ; arg[1] = rate ; arg[2] = time ;
            invocation.setArgument(args);
       LogAroundAdvice advice = new LogAroundAdvice();
      (k) Object val = advice.invoke(invocation); (d)
    }   return val; (l)
}
```

ClientApp

```
                                    (a)                    ← generates proxy class object.
IntrAmtCalculator proxy = ctx.getBean("pfb", --);
Float result = proxy.calcIntAmt(1000,2,20);
(m)                    (b)
```

Note While working with AOP our target class must not be final class and target methods must not be static, final because it gives problem towards generating proxy class by extending from target class and overriding target class methods in proxy class.

10/08/15

Note It is always recomanded to take seperate class for every cross cutting logic/ secondary logic. It is better not to mix up multiple logics in a single class.

⇒ The best usecases for AroundAdvice are caching, logging, performance monitoring.

⇒ Cache/buffer is a temporary memory that holds data for temporary period.

⇒ In client-server application, the cache at client side holds server application results and uses it across the multiple same request to reduce the network round trips bit client & server

⇒ In AOP programming, we store target method result in cache and we use that cache across the multiple same method calls. we generally take either arrays (or) collections (Hashmap) as cache.

Understanding flow of execution related to cache advice

HashMap object (cache)

calcIntAmt(8000,2,20)	1600

```
@Override
public Object invoke(MethodInvocation invocation) throws Throwable{
    String key = invocation.getMethod.getName() + Arrays.toString(invocation.getArgument
    Object val = null;
    if(!cache.containsKey(key)){
        val = invocation.proceed();
        cache.put(key, val);
    S.o.p("From target method");
    return val;
```

NOTE While configuring cacheAspect / Advice along with other advice / aspects, it will recomanded to configure them after cache advice because if target method is getting its result from cache there is no need of performing logging / performance monitoring.

11/08/15

Before Advice

⇒ The interface ie, implemented by target class is called "Proxy Interface". It actually called as Proxy Interface because the dynamically generated proxy class implements this interface. We need to configure this interface while configure "ProxyFactoryBean" class in xml file.

⇒ Once proxy interface is configured in xml file the JDK libraries will be used to generate the proxy class dynamically at runtime and this generated proxy class does not extend from target class rather implements proxy interfaces, Due to this here we can take target class as final class and target methods as final methods.

NOTE

⇒ When proxy interfaces is not configure the CGLIB libraries will be used to generate proxy class extending from target class.

For example on proxy interface refer AOP Proj2.

2) Before Advice

⇒ This Advice executes before the beginning the execution of target method and once Advice method is executes the control authentication goes to target method. After completing the execution of target method does not come back to Before Advice method.

⇒ To develop Before Advice our class must implement method BeforeAdvice (Interface) should provide implementation to before(-, -, -) method.

sample code

```
public class MyBeforeAdvice implements MethodBeforeAdvice
{
    public void before(Method method, Object args[], Object target)
    {                          ↦ To get          ↦ To access and      ↦ To access target class object
                               Target method info   modify target
      ---                                           method arguments
      ---
    }
}
```

⇒ The use cases of Before Advice are Auditing, security check and etc.

The important/control points of Before Advice

a) Can access and modify target method arguments

b) Can't access and modify target method return value because the control does not come back Before Advice once the target method execution is completed.

c) Can not control target method execution begin from Advice, but can stop/abort target method execution by throwing exception in before(-,-,-) method.

⇒ Logging keeps track of various components that are involved in the flow of execution of application.

⇒ Auditing keeps track of various activities that are taken place in the execution of the application.

⇒ Generally the audit information will be written to audit log file or audit table. It is like writing username, ordered time and etc. to audit file

AOPProg3 (Before Advice-Auditing)

```
|-> src
    |-> com.nt.cfgs
        |-> applicationContext.xml
    |-> com.nt.aspect
        |-> AuditingAdvice.java
    |-> com.nt.service
        |-> OrderApprover.java
|-> test
    |-> ClientApp.java
```

13/08/15

⇒ Security means authentication & authorization.

⇒ checking the identity of a user is called authentication. checking its access permissions of a user is called authorization. Security is main authentication.

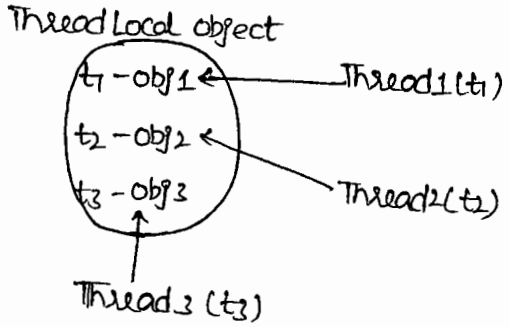
⇒ We need to authenticate the user before entering in to target method / business method. So it is recomanded to develop security logic as Before Advice logic.

⇒ Since target method call cannot supply username, password every time we need to make client performing login operation submitting username, password before calling target method. These username, passwords must be saved temporarily to use them when real business method / target method is called. For this we can use ThreadLocal object.

⇒ If we want to make data specific to each thread having global visibility without using local variables and synchronization concepts then we can keep that data in ThreadLocal object that is taken as instance variable. This ThreadLocal object allows each thread to keep one object and does not allows other threads to access and use that object. So we can say ThreadLocal object maintaining data having Thread scope.

⇒ In java we can keep data in 4 scopes.

1) local
2) instance
3) class
4) thread scope



Example

userlist (db table in oracle)

uname (vc2)	pwd (vc2)
raja	rani
king	kingdom

AOPProj4 (Spring AOP Decl - BeforeAdvice - Security)

```
|→ src
    |→ com.nt.cfgs
        |→ applicationContext.xml
    |→ com.nt.dao
        |→ AuthenticateDAO.java
    |→ com.nt.aspect
        |→ AuthenticationManager.java (helper class)
        |→ SecurityAdvice.java (Before Advice)
        |→ UserDetails.java
    |→ com.nt.service
        |→ IntrAmtCalculator.java
    |→ test
        |→ ClientApp.java
```

jar files : AOP LIB + spring-jdbc-<ver>.jar + spring-tx-<ver>.jar
+ commons-dbcp.jar + commons-pool.jar + ojdbc14.jar

14-08-15

3) After Advice / After Returning Advice

⇒ This Advice Method/logic executes once target method finishes the execution and before it returns the control to caller method. This Advice can catch the return value but can't modify the return value. To develop this advice we need to take a class implementing AfterReturningAdvice (Interface) and should provide implementation for afterReturning(-,-,-) method

```
public class MyAdvice implements AfterReturningAdvice
{
    public void afterReturning(Object retVal, Method method, Object[] args, Object target)
    {
        --
    }
}
```

Use Cases

1) Generating Discount voucher for next purchase based on the current purchase bill amount
2) ATM pin Generator

Important Points / Control Points

1) we can access target method arguments but there is no use of modifying them because control comes to advice method only after finishing the execution of target method.
2) Can not control target method execution because control comes to advice method only after executing target method. But by throwing exception we can stop target method sending them value to caller method.
3) Can access the return value but can not modify the return value ie, being returned.

17/08/15

AOPProj5 (Spring AOP Declarative Approach)

```
|-> src
    |-> com.nt.cfgs
        |-> applicationContext.xml
    |-> com.nt.service
        |-> shoppingStore.java
    |-> com.nt.aspect
        |-> DiscountCoupon.java
    |-> test
        |-> ClientApp.java
```

Jar files: AOP Library jar files + com.springsource.org.aopalliance-1.0.0.jar

⇒We can use afteradvice to check wheather the return value generated by the target method is valid value or not. If not valid we can throw exception in after advice and we can stop that return value being return value true in ClientApp.

Usecase: If target method creates ATM Pin we can check whether that generated pin is weak pin or strong pin in after advice if it is weak pin then we can throw an exception.

AOP Prog 6 (Spring AOP - Declarative - After Advice - pinverifier)

```
|→ src
    |→ com.nt.cfgs
        |→ applicationContext.xml
    |→ com.nt.service
        |→ PinGenerator.java
    |→ com.nt.aspect
        |→ CheckPinAdvice.java
    |→ test
        |→ ClientApp.java
```

Jar files: Same.

18/08/15

4) Throws Advice

⇒ This Advice executes when exception is raised in target method. using this advice we cannot catch and handle the exception but we can see the details of exception that is raised.

⇒ We can use this advice to develop common/central ExceptionLogger that logs all the exception related messages that are raised in various target methods of target class.

⇒ We can catch and handle the exception raised by target method in the client Appln where target method will be called.

⇒ To develop this advice we need to take class that implements "ThrowsAdvice(Interface)" (Marker → Empty interface) and we need to place 1 or more afterThrowing methods having the following signatures.

```
public void afterThrowing(<Sub class of Throwable>)
public void afterThrowing(<subclass of Throwable>, Method method,
                          Object args[], Object target, <subclass of Throwable>)
```

control points / important points

⇒ We can access target method parameters, but there is no use of modifying them.

⇒ We can not control target method execution because control comes to throws advice after executing target method that to for exception raising.

⇒ We can not catch the return value of target method.

AOPProj7

```
|——→ src
        |——→ com.nt.cfgs
                |——→ applicationContext.xml
        |——→ com.nt.aspect
                |——→ ExceptionLoggerAdvice.java
        |——→ com.nt.service
                |——→ PriceCalculator.java
        |——→ test
                |——→ ClientApp.java
```

Jars : same

⇒ If Exception is raised in Target method control comes to throws Advice and looks for afterThrowing method in the following order,

a) Exact exception type
b) super class of raised Exception.

Note If multiple afterThrowing method contains exact method match then more parameters method executes.

Note If multiple afterThrowing method contains super class of raised exception then more parameters method executes.

Note If 1 afterThrowing() contains Exact match and another afterThrowing contains super type then exact match afterThrowing() executes.

⇒ We can also use throws advice to wrap (or) translate one form of exception to another form of exception.

```
public void afterThrowing(IllegalArgumentException ie) {
    throw new NumberFormatException("Exception Type changed here");
    }
```

Generated Proxy class

```
public class PriceCalculator$Proxy extends PriceCalculator
  {
    public float calcBillAmt(float price, int qty) {
      float result = 0.0f;
      try{
        result = super.calcBillAmt(price, qty);
        }
      catch(IllegalArgumentException iae){
      ExceptionLoggerAdvice advice = new ExceptionLoggerAdvice();
          advice.afterThrowing(iae);
        }
      return result;
  }
```

Advice type	When executes advice logic	Access to target method params	Access to target method return value	control on target method execution
1) Around Advice	around target method execution	can see and modify	can see and modify	Yes
2) Before Advice	before target method execution	can see and modify	NA	NA (can stop by throwing exception)
3) After Advice	after target method execution but before it returns the control to caller	can see but can not modify	can see but cannot modify	NA
4) Throws Advice	when exception is raised in target method	can see but cannot modify	NA	NA

19/08/15

b) Programmatic Spring AOP:

⇒ Here no xml files support or configurations support will be taken to generate Proxy class by supplying target class, advice class. All instructions will be given directly in client Application through java statements.

⇒ This is not recommanded Approach

⇒ Here we use Proxyfactory class directly in clientApp to generate proxy class.

code for this

```
//Get Proxyfactory class object
Proxyfactory pfactory = new Proxyfactory();
// set Target class object
pfactory.setTarget(--);
//set Advice class object
Pfactory.addAdvice(--);
// get proxy class obj
Target proxy = pfactory.getProxy();
// call target method
proxy.bm1(); proxy.bm2();
```

bm1() → business method
bm2() ↗

Spring AOP Proj 8 (programmatic Approach)

```
|-> src
    |-> com.nt.aspect
        |-> PerformanceMonitoringAdvice.java
    |-> com.nt.service
        |-> ArithmeticService.java
    |-> test
        |-> ClientApp.java
```

Jar files : same AOP lib jar files

Point Cut

⇒ So far the aspects that we are developing will be applied on all the methods of target class. In some situations we do not want to apply aspects on certain join points on methods of target class for this we can write additional conditional logic in advice class to execute advice logic only for specific (target methods) join points (target methods).

code

```
public class PerformanceMonitoringAdvice implements MethodInterceptor{
    public Object invoke(MethodInvocation invocation) throws Throwable{
        if( invocation.getMethod().getName().equals("add")){
            //advice logic will be applied only on "add" target method.
            ---
        }
        ---
        ---
}
```

⇒ In the above approach, the generated Proxy class override all the methods of target class having logic to apply aspect on target methods. But advice logic will be executed only for "add" method because of the condition that is specified.

⇒ This indicates Proxy class is not generated in optimised manner because it is also overriding those target methods on which we do not want to apply aspects moreover those methods of target class should be executed directly without touching proxy class when they are called from client Application.

⇒ To over this problem use Point Cut.

PointCut

⇒ PointCut is a collection of JoinPoint on whom we want to specify the advice. It specifies which aspect has to be apply on which join point.

⇒ There are 2 types of point cuts

1) static point cut Allows to specify target class name, method names while preparing the point cut but we can not specify target methods argument values. This indicates static point cuts will be processed during weaving process that generates proxy class.

2) Dynamic PointCut

-> This allows to specify target class, target method names and their argument values that means this pointcut will be processed after target method call on proxy object dynamically at runtime.

Ex: Apply aspect on "add" method is static pointcut.

Ex: Apply aspect on "add" method when argument values are less than 100 is dynamic point cut.

⇒ While working with Pointcut the generated proxy class overrides only methods that are specified in pointcut to apply aspects. Due to this the proxy class code will be optimised to improve the perfoimance.

⇒ To develop user defined point cut we need to take a class implementing PointCut(I).

⇒ Spring api has supplied some predefined classes implementing this interface.

Static PointCuts classes

⇒ StaticMethodMatcherPointCut

⇒ NameMathMethodPointcut

⇒ JdkRegularexpMethodPointcut

Dynamic PointCuts classes

⇒ DynamicMethodMatcherPointcut (abstract class)

Note

* To develop custom pointcut classes extend from above classes instead of implementing pointcut interface directly.

⇒ Advisor is the combination of pointcut + aspect/advice. In order to apply aspects specifying pointcut we take the support of advisor. every Advisor class is the implementation of Advisor (Interface). The spring api supplied predefined advisors classes are DefaultBeanFactoryPointcutAdvisor, NameMatchMethodPointcutAdvisor.

Static Pointcut

⇒ While working with static pointcut classes we need to override matches() method that give access to target class and target method. If this method returns true for certain target method aspect will be applied for that method otherwise it will not be applied.

For example appln on programmatic spring AOP having programmatic pointcuts.

Spring AOP Proj8 (Programmatic-pointcut)

20/08/15

Dynamic Pointcut

⇒ In order to develop Dynamic Pointcut we need to take a class extending from DynamicMethodMatcherPointCut class and implements matches(Method method, class<?>target, object[] args) method.

⇒ If this method returns true for certain target method then aspect will be applied on that method otherwise aspect will not be applied.

For Example

Spring AOP Proj8 (Programmatic-pointcuts)

⇒ Instead of developing seperate user defined class by extending from the abstract class staticMethodMatcherPointcut / DynamicMethodMatcherPointcut we can use spring with api supplied concrete static pointcut class NameMatchMethodPointcut as shown below,

```
NameMatchMethodPointCut nmp = new NameMatchMethodPointcut();
nmp.setMappedName("add","mul");
```

```
DefaultPointcutAdvisor advisor = new DefaultPointcutAdvisor(nmp, new PerformanceMonitoringPointcut());
pf.addAdvisor(advisor);
```

Working with Pointcuts in declarative spring AOP

⇒ In this approach, we can configure pointcut classes, Advisor classes as beans in configuration file and we can specify advisors directly as the value of interceptor names property while configuring proxy factory bean.

⇒ Instead of working with NameMatchMethodPointcut, DefaultMethodAdvisor PointCut seperately we can use NameMatchMethodPointcutAdvisor class directly.

Ex: AOPProj9 (Declarative-Pointcuts)

```
|→ src
    |→ com.nt.cfgs
        |→ applicationContext.xml
    |→ com.nt.aspect
        |→ PerformanceMonitoringAdvice.java
        |→ LogAroundAdvice.java
        |→ CacheAdvice.java
    |→ com.nt.service
        |→ ArthemeticSevice.java
    |→ com.nt.pointcut
        |→ MyDynamicPointcut.java
    |→ test
        |→ ClientApp.java
```

Jars: same

⇒ We can develop static pointcut either as user defined class extending from StaticMethodMatcherPointcut or) we can use spring supplied direct concrete classes like NameMatchMethodPointcut.

⇒ In order to work with dynamic pointcut we must take userdefined class extending from DynamicMethodMatcherPointcut and there are no direct concrete classes supplied by spring api to work with dynamic pointcuts.

21/08/2015

AspectJAOP:-

⇒ A third party AOP framework where we can develop aspect classes as POJO classes with out having tight coupling with spring api. i.e., aspect classes need not implement spring api interfaces and need not extend from spring api classes. but we need aspectj api inside the aspect classes.

Q what is the difference between SpringAOP and AspectJAOP?

SpringAOP	PlainAspectJAOP
⇒ Performs runtime weaving	⇒ Performs compile time weaving
⇒ Generates Proxy class at runtime	⇒ Generates the proxy class at compiletime uses AspectJ compiler
⇒ Supports only method as Joinpoints	⇒ Supports fields, constructors, methods as joinpoints
⇒ Supports both static and dynamic pointcuts	⇒ Supports only static point cut
⇒ Does not allow annotations	⇒ Allows annotations
⇒ can be used in programatic and declarative (xml)	⇒ can be used in declarative (xml) and through annotations.

⇒

* From Spring 2.0 the AspectJAOP is integrated with spring AOP due to this some limitations are applied on AspectJAOP.

i) Supports only method level join points
ii) performs runtime weaving ie, proxy class will be generated at runtime.
iii) still supports only static point cuts
iv) NO AspectJ compiler is required.

⇒ We can use Spring integrated AspectJ AOP in two ways.

a) Declarative AspectJ AOP (using xml)

b) Annotation AspectJ AOP.

Note AspectJ AOP also supports four types of Advices.

a) Declarative AspectJ AOP

⇒ Here we develop Aspect classes as POJO class by using AspectJ API internally and we configure these classes in xml file by using tags as Aspect classes.

⇒ we need to import AOP name space for these configuration. The important tags are

<aop:config> → To enable AspectJ Aop from spring container
<aop:aspect> → To configure spring bean as Aspect class
<aop:around> → To configure Aspect class method as around advice
<aop:before> → To configure aspect class method as before advice
<aop:after-returning> → To configure aspect class as after advice
<aop:pointcut> → for pointcut configuration

⇒ we use OGNL Syntax for point cut expression

Syntax

execution (<return type> <package name> <class name> <method name> (--))

1) Around Advice

⇒ This advice executes around target methods.

usecases caching, logging, performance Monitoring and etc..

⇒ To develop this advice we can use any arbitary class having arbitary method with the following signature.

public object <method> (ProceedingJoinPoint PJP) throws Throwable.

⇒ ProceedingJoinPoint PJP can be used to get target class, method information and to control target method invocation.

Notes

⇒ while working with AspectJ AOP Around advice the modifications done in target method in around in Advice will be reflected only when we call pjp.proceed() method having that object array as argument. Here object array holds the modified argument value.

⇒ When we run client Application of spring Integrated AspectJ AOP the internally created IOC container maintains in memory meta data along with AOP configuration based on <aop:config> and its sub tags. when ctx.getBean() method is called with the target class bean id. The IOC container does not create target class object because of <aop:config> tag based configurations. It generates proxy class extending from target class and returns that proxy class object back to client application.

⇒ Like spring AOP if we start working with proxy interface (the instance that is implemented by target class) in AspectJAop the generated proxy class will come because of JDK libraries for this we need to specify proxy interface name in OGNL expression of pointcut. In this AspectJ AOP if proxy interfaces are not specified then CGLIB libraries will be utilize to generate proxy class.

Example

AspectJAop1 (Around Advice)

```
|-> src
    |->com.nt.aspect
        |-> LogAroundAdvice.java
        |-> CacheAdvice.java
        |-> PerformanceMonitorAdvice.java
    |->com.nt.service
        |->IntrAmtCalculator.java
    |-> com.nt.cfgs
        |-> ApplicationContext.java
    |-> test
        |-> ClientApp.java
```

Jar files

AOPJars + aspectjrt-1.5.3.jar + aspectweaver-1.5.3.jar

2) Before Advice

⇒ This advice executes before executing target method. using this advice we can access target method argument we cannot control target method execution and we cannot catch return value.

⇒ using this Advice we can stop target method execution by throwing exception.

To develop this advice we can take any class having any method with following signature.

```
public void <method> (JoinPoint jp)
                (or)
public void <method> (JoinPoint jp, param1, param2, ..)
                                    └> To hold target method argument values.
```

⇒ Join Point object just gives target method details where as proceeding Join point object gives target method details and control on target method invocation.

⇒ Before Advice usecases are security check, Auditing, .. and etc..

24/08/15

Aspect JAOP2(Before Advice)

```
|-> src
    |-> com.nt.cfgs
        |-> applicationContext.xml
    |-> com.nt.aspect
        |-> AuditingAdvice.java
    |-> com.nt.service
        |-> OrderApprover.java
|-> test
    |-> ClientApp.java
```

⇒ In AspectJAop the before Advice can access target method arguments but cannot modify their values. It is not similar to Spring AOP Before Advice.

⇒ while working with Before Advice we can even get target method argument with out taking support of JoinPoint object we need to take additional parameters support in advice method to store the target method argument and we must specify those parameters in point cut expression.

```
public class AuditingAdvice {
    public void audit (int orderId |JoinPoint jp){
    --
    }
}
```

In applicationcontext.xml

```
<aop:pointcut id="ptc1" expression="execution(*.com.nt.service.*.*(--) and
                                    args(orderId))" />
```

⇒ For AspectJAop with Before Advice performing security check refer page NO: 23 in AOP booklet 2.

⇒ while working with AspectJAOP If we call ctx.getBean() method by using the bean id whose class name is specified in point cut ONGL expression then we get proxy class object otherwise we will get given bean id object.

25/08/15

3) After Advice / AfterReturning Advice

⇒ This advice method executes when target method completes the execution and before it returns the control/value to caller.

⇒ To develop this advice we can take any class having the following signature method

public void <method> (JoinPoint jp, <param to hold return value>)
↳ To get target method details.

⇒ Usecases are : 1) Generating discount coupon for next purchase
2) ATM Pin verifier

⇒ This advice allows to access target method arg values.

⇒ This advice allows to access return value but does not allow to modify.

⇒ By throwing exception in this advice we can stop return value being returned to caller method.

⇒ The Parameter taken in advice method to hold return value must be configured in <aop: AfterReturning> by using "returning" attribute.

<aop: afterReturning method = "cupon" pointcut-ref = "ptc1" returning = "billAmt" />
↳ advice method parameter name to hold the generated return value.

AspectJAOP5 (AfterAdvice - Discount Coupon)

- src
  - com.nt.cfgs
    - applicationContext.xml
  - com.nt.source
    - shoppingstore.java
  - com.nt.aspect
    - DiscountCouponAdvice.java
  - test
    - ClientApp.java

4) Throws Advice

⇒ This advice method executes if exception is raised in target method.

⇒ This advice is useful to develop central, common Exception logger.

⇒ This advice does not catch and handle the exception, but allows to see the details and to translate one form of exception to another form of exception.

⇒ To develop this advice we can take a class having method with following signature.

public void <method name> (JoinPoint jp, <param of type Subclass of Throwable>)

⇒ The additional param of this advice method that is capable of reffering the raised exception must be specified in <aop: after-throwing> tag by using "throwing" attribute

<aop: after-throwing method="--" pointcut-ref="--" throwing="--" /> (throwing: parameter name of throws advice method that is capable of reffering exception raised in target method.)

⇒ Throws Advice is even useful for expression graphics. It is transforming one form of exception to another form of exception.

Annotations driven AspectJAOP

⇒ Here we donot use xml file to configure advice class and advice methods rather we use list of annotations given by AspectJAop @aspect → to configure class as Aspect class

@ Around → to configure method as around advice

@ Before → to configure method as before advice

@ AfterReturning →

@ AfterThrowing →

@ Pointcut → To specify pointcut expression with reusability

To make @Ioc containa recognise all these annotation we need to place <aop: aspectj-autoproxy> tag in xml file.

26/08/15

⇒ While working with AspectJAop we can take the advice classes as POJO classes and we can place multiple advice logics in a single class.

⇒ @Around, @AfterReturning, @Before, @AfterThrowing annotations to place pointcut expressions directly.

⇒ To place pointcut expression with reusability we can use @Pointcut.

Aspect JAnnoAPP

```
|-> src
    |->com.nt.cfgs
        |->applicationContext.xml
    |-> com.nt.aspect
        |-> LogAroundAdvice.java
        |-> CacheAdvice.java
        |-> PerformanceMonitoring.java
    |-> com.nt.service
        |-> IntrAmtCalculator.java
    |-> test
        |-> clientApp.java
```

Refer page 29 AOP Section

⇒ While working with AspectJAop with annotations the order they follow to apply annotation advices on target method will be decided based on the order that is used to configure advice classes as spring beans in applicationContext.xml.

⇒ If advice classes are not configured in xml file and if they are configured through annotation @component then we can defined the @order Annotation.

⇒ @component or @service can be used to configure ordinary java class as springbean

⇒ @Aspect cannot be used to configure ordinary java class as Aspect class but it can be used to configure spring bean as Aspect class.

⇒ While working with annotations based AspectJAop <aop:aspectj-autoproxy/> is alternate <aop:config> tag to enable AspectJAop based on the annotations.

⇒ Throws advice not only executes for the exception that is raised in target method. It is also executes for the exception that is raised in advices that are related to target method.

⇒ when we defined multiple advice methods in single aspect class having same pointcut expression then instead of defining that pointcut expression seperately for every method it is recommanded to take dummy method name in same aspect class having pointcut expression defined through @pointcut and specifying that dummy method name in every advice method as pointcut expression.

```
public class LoggingAspect {
    @Pointcut ("execution (* com.nt.service.Math.*(..))")
     public void mypointcut(){ }
    @Around ("mypointcut()")
   public object log (ProceedJoinPoint Point pjp) throw Throwable {
    ---
    }
```

⇒ For example application on above concept refer :31 of AOP.

```
@pointcut("execution (* com.nt.service.Math.*(..))")
    public void mypointcut() { }
```

⇒ Dummy method having pointcut expression for reusability.

⇒ AOP is designed on decorative, proxy design patterns.

27/8/15

Transaction Management

⇒ The process of combining related operations into single unit and executing them by applying do everything or nothing principle is called Transaction Management (TM).

⇒ The Withdraw Amount from source account, deposit amount into Destination account must happen in Transactional environment.

⇒ In Employee registration employee details insertion into HR DB and into Payroll DB must happen in Transactional environment.

⇒ Transaction management allows to implement ACID properties.

A → Atomocity
C → Consistency
I → Isolation
D → Durability

⇒ The process of combining indivisible operations in to single unit is called "Atomocity". Atomocity word is derived from the term called "Atom".

⇒ Even though the rules kept on database s/w (Office balance must not be -ve) are violated during the course of Transaction, but if there is a guaranty that no rule will be violated at the end of Transaction, then we can say Database is consistent.

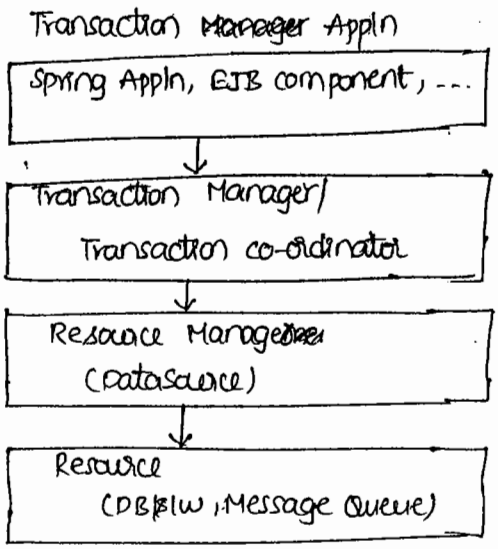
⇒ Applying locks on DB s/w and allowing 1 user / 1 application at a time to manipulate DB data is called Isolation

⇒ Bringing DB back to normal state by using log files, backup file even though it is crashed indicates durability of DB.

⇒ Every Tx management contains 3 operations.

```
Begin Tx
  ---
  ---
    continue Tx (Execute logics)
  ----
  ----
commit Tx / Rollback Tx
```

Tx Architecture



⇒ Transaction resource must allow data commit or rollback operation.

⇒ File cannot taken as Tx Resource. DB s/w, Message Queues can be taken as Tx Resource.

⇒ Based on the Tx Appln instruction the Tx manager connects to Resource through Resource manager to commit/ Rollback Tx.

⇒ Based on no.of resource that are involved in a transaction there are 2 types of Transactions

a) Local Transaction

Single resource will participate in a Transaction boundary

eg: Transfer money operation b/w two accounts of same bank.

b) Global Transaction (XA Tx)

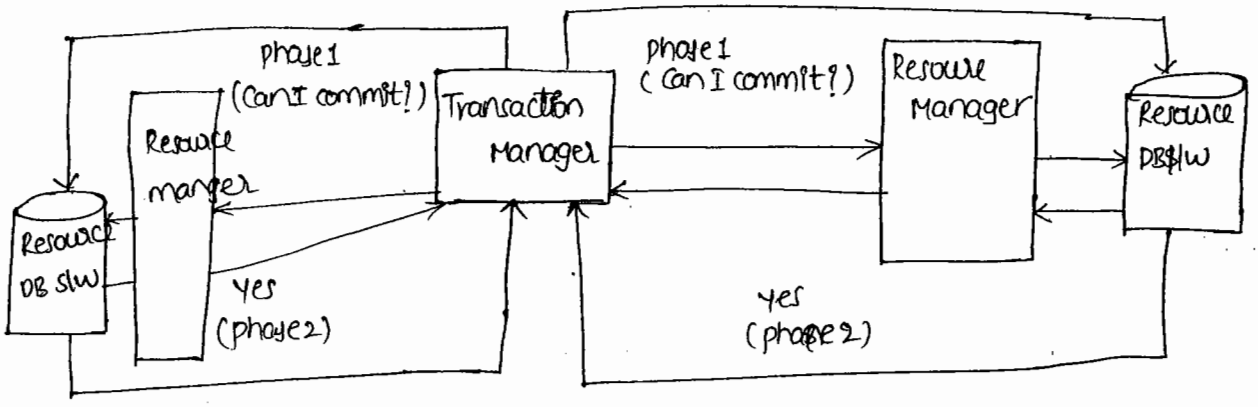
Multiple resources will participate in a Transaction boundary. Here commit or rollback takes place on multiple resources in a single shot.

eg: Transfer money operation b/w two accounts of two different banks.

⇒ Global Tx runs based on 2pc protocol (2 phase commit)

phase 1 Tx manager talks with all the resources through Resource manager seeking permission to commit the Tx by checking the data consistency.

phase2 If all resource managers responds and says yes/ok then the data of multiple resources will be committed, if any resource says "no" then the data of multiple resources will be rolled back.

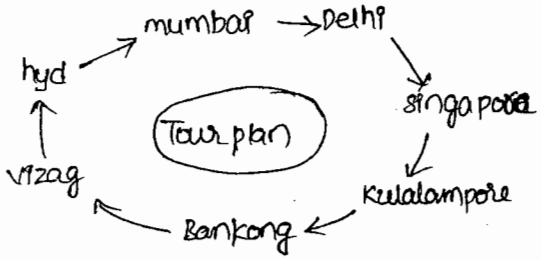


Eg: JDBC, hibernate supports only local Tx.

Spring, EJB supports both Local & Global Txs.

Transaction Models

1) Flat transaction
2) Nested transaction



⇒ When the above Tour plan is given to flat Transaction model application if one journey tickets are not available it will cancel the remaining journey tickets because all the bookings can be taken as direct operations can be taken as MainTxappln.

⇒ If the above Tour plan is given to Nested Transaction it will confirm remaining journey tickets even though one or another journey tickets are not available. Because here each journey ticket booking will be taken as sub transaction of Main transaction so the success/failure on one transaction does not support on another transaction.

Flat Tx

```
Main Tx {
  journey1 ticket booking;
  !
  journeyn ticket booking;
}
```

Nested Tx

```
Main Tx {
SubTx1 {
    journey1 ticket booking;
  }
!
SubTxn { journeyn ticket booking;
}
```

⇒ JDBC, Hibernate, EJB supports only Flat Tx.

⇒ Spring supports both Flat Tx and Nested Tx.

28/08/15

⇒ Generally we use JDBC code for Local Transaction management as shown below,

Local Tx sample code using JDBC

```
try {
    con.setAutoCommit(false); // Begin Tx
    ---
    ---   } Tx logic
    ---
    con.commit();
  }
 catch (Exception e) {
    con.rollback();
  }
}
```

⇒ To perform Distributed Tx management we need either Application server (JEE server) or Third Party api like "atomokis" that gives UserTransaction object as Tx manager. We also need XA jdbc driver, XA DataSources to get Tx manager required for Distributed Tx. Here we never directly call con.commit() (or) con.rollback() directly we ask Tx manager to perform these operations by getting con through Resource Manager (DataSource). The servers like Weblogic to supply "UserTransaction" object in its Jndi Registry having fixed Jndi name like "javax/transaction/UserTransaction".

⇒ We use JTA Support to perform Global Tx mgmt in a programmatic Approach. We use EJB to perform Global Tx mgmt in a declarative approach (xml cfgs/annotation cfgs).

⇒ In programmatic Approach we write Tx code directly inside the Business method. So disabling Tx service is very complex.

In Declarative Approach Business methods contains only business logic But the Tx cfgs will be placed in xml file. The container/server/framework enable Transaction service on business methods based on xml file cfgs.

⇒ JDBC, JTA supports only Programmatic Tx mgmt.

⇒ EJB supports both programmatic, Declarative Tx mgmt.

Global Tx in the component that is deployable in the Sewer using JTA (Programmatic Approach)

```
InitialContext ic = new InitialContext();
UserTransaction ut = ic.lookup ("javax| transaction| UserTransaction");
                                  ⟼ jndi registry ⟻
try {
      ut.begin();
      ---   || Tx logic
      ut.commit();
    }
catch(Exception e){
      ut.rollback();
    }
```

Declarative Approach Tx mgmt using EJB

EJB Component

```
public class MyComponent implements SessionBean {
      public void bm1(){
        ---  //only business logic and no Transaction logic
      }
}
```

ejb-jar.xml

→ class cfgs
→ tx cfgs

⇒ Based on xml cfgs the EJB contains applied Tx service on the methods of EJB components.

⇒ The EJB's declarative Tx mgmt service is really nice but it is heavy weight.

Spring Transaction

⇒ In the above discussion we need to use different java, JEE api's we need another api's like hibernate, JDA etc.. to perform different modes of Tx management.

⇒ Due to this switching to local Tx to distributed Tx on switching from one tx implementation to another tx Tx implementation becomes complex for programmer.

⇒ Spring provides unified env to work with any Tx because its a framework providing abstraction layer on multiple technologies. Due to this we can make a

spring to use internally any implementation of tx but we can write unified code for tx mgmt.

⇒ Due to this switching to distributed tx from local tx, switching from one Tx impln to another Tx impln does not make programmer to change spring code much.

⇒ Spring supports 4 ways of Tx mgmt.

a) Programmatic Tx Management
b) Declarative Tx Mgmt using Spring AOP
c) Declarative Tx Mgmt using Aspect AOP
d) Annotations based Tx management using AspectJ AOP

⇒ In spring transaction service is secondary logic/ cross cutting logic concern, so it is given as a pre-defined Aspect logic as the combination of Around advice and throw Advice type.

⇒ Tx will begin while entering to method and will be committed when method returns a value or will be rolled back when method throws exception so the Tx service is Around Advice, throws Advice.

30/8/15

⇒ In order to perform Tx mgmt we need to choose the appropriate Tx manager/Tx coordinator

DataSourceTransactionManager → for underlying JDBC (Local Tx)
HibernateTransactionManager → for hibernate code (Local Tx)
JtaTransactionManager → for distributed Tx.

---

b) Declarative Tx Mgmt using spring AOP

1) TransactionProxyFactoryBean → Generates proxy class applying Tx service on the target methods.

2) NameMatchTransactionAttributeSource → Allows to specify Tx attribute on target methods. It is like pointcut.

Note To create DataSourceTransactionManager the "dataSource" object is dependent.

⇒ In programmatic Tx Mgmt we write code inside the business method which is against of AOP, so do not use programmatic Tx Mgmt

⇒ In Declarative Tx mgmt we write Tx related cfg in xml file so target method just contains business logic but Tx service will be applied on Business methods because of Proxy class.

Tx Mgmt Prof 1 (Local Tx - Decl Spring - AOP)

```
|-> src
    |-> com.nt.cfgs
        |-> applicationContext.xml
    |-> com.nt.service
        |-> BankService.java
    |-> com.nt.dao
        |-> BankDAO.java
        |-> BankDAOImpl.java
    |-> com.nt.test
        |-> ClientApp.java
```

jars : AOP Lib + SPJDBC Lib

⇒ In Declarative Tx Mgmt, the Tx manager rollbacks the transaction automatically if unchecked exception is raised in the target method (runtime exception and its sub classes) otherwise Tx will be performed.

Note

⇒ In Declarative Tx Mgmt we can specify Tx attribute on the target method to make target method running with no transaction (or) new transaction on client supplied tx.

Internals in Prof1

targetMethod/serviceMethod → transferMoney

DataSourceTransactionManager

NameMatchTransactionAttributeSource → specifies the tx attributes on targetmethod.

PROPAGATION_REQUIRED → transaction attribute

TransactionProxyFactoryBean → generates proxy class having Tx service

RuntimeException → When this exception is raised the tx manager rollbacks the Tx.

Transaction Attributes

1) Required (PROPOGATION_REQUIRED)
2) Requires New (PROPOGATION_REQUIRES_NEW)
3) Supports (PROPOGATION_SUPPORTS) (Default)
4) NEVER (PROPOGATION_NEVER)
5) NotSupported (PROPOGATION_NOT_SUPPORTED)
6) Mandatory (PROPOGATION_MANDATORY)

1) Required

⇒ If client Appln calls the Business method in a Tx then business method runs that Tx

⇒ If client Appln calls the b.method with out Tx the b.method runs in a New Tx.

Note : It is most popularly used Tx attribute.

2) Requires New

⇒ b.method/Target method always runs in a new Tx irrespective of weather client App calls b.method in a Tx (or) not

3) Supports

⇒ B.methods runs in a client Appln Tx if client Appln calls the b.method in a Tx.

⇒ B.method run with out Tx, if client appln calls the b.methods with out Tx.

4) Never

⇒ Client Appln must not call b.method in a Tx, if called Exception will be raised.

5) Mandatory

⇒ Client Appln must call b.method in a Tx, if not called Exception will be raised.

6) Not Supported

⇒ B.method always run with out Tx irrespective of weather client appln has called b.method in a Tx or not.

Transaction Attribute	client's transaction	Beans transaction
1) Required	none	T2 (New Tx)
	T1	T1
2) Requires New	none	T2 (New Tx)
	T1	T2 (New Tx)
3) Supports	none	none
	T1	T1
4) Mandatory	none	error
	T1	T1
5) NotSupported	none	none
	T1	none
6) Never	none	none
	T1	error

18/15

Declarative Tx mgmt using AspectJAOP

⇒It is most regularly use Tx mgmt mechanism. To work with this we need to import AOP, tx namespaces. In order to activate Tx service as advice with bean id to spring Tx attributes on target methods we can use <tx:advice> <tx:attributes>, <tx:method> tags. The Tx advice performs commit (or) roll back operations through Tx manager. so we need to specify Tx manager while cfg Tx advice.

⇒In order to apply Tx advice on the specific methods of Target service class use <aop:config>, <aop:advisor>, <aop:pointcut> tags as shown below.

Sample code

```
<bean id="txmgmr" class="org.sf.jdbc.dataSource.DataSourceTransaction">
    <property name="dataSource" ref="dbcpds"/>
</bean>
<tx:advice id="txAdvice" transaction-manager="txmgmr">
    ↳ represents tx Advice
<tx:attributes>
<tx:method name="transferMoney" propagation="REQUIRED" read-only="false"/>
    ↳ target method        ↳ Transaction Attribute
</tx:attributes>
</tx:advice>

<

<aop:config>
<aop:pointcut id="ptc1" expression="execution(* com.nt.service.BankService.
                                              TransferMoney(--))";
<aop:advisor advice-ref="txAdvice" pointcut-ref="ptc1"/>
    ↳ Apply Tx service on the methods specifying in pointcut expression
</aop:config>
```

Ex Refer Appln on AspectJAOP based Declarative Tx mgmt the refer App 2 of page no:35 of AOP.

Note If we specify star it represents all the methods of target class

```
<Tx:method name="*" propagation="REQUIRED"/>
```

Note In Declarative, Annotation based Tx mgmt the Transaction manager will rollback the transaction only when target method throws unchecked Exception(runtime exception in subclass) does not rollback for checked Exception.

Note we generally design DAO class one per Table strategy. Instead of applying Tx service on DAO class methods directly use DAO classes in service class methods and apply Tx service on that methods.

Annotations driven AspectJ AOP Tx mgmt

⇒ For this transaction the Tx configurations will not be done in xml file they will be configured directly on Target class methods by using @Transactional to make IOC container detecting and using these Annotations we can use

<Tx:annotations-driven transaction-manager "--" />

while working with @Transactional we can specify read only param and propagation param(for transaction attributes)

Note In order to apply same Tx attribute on multiple methods (or) Target class we can place @Transactional on the top of Target class.

Note other use cases of Tx mgmt, in Employee registration this detail should be in HR DB, finance DB.

⇒ In student Registration, student details should be saved in Library DB and Admissions DB. In online shopping goods delivery and payment must be executed in a Tx.

11/9/15

Global Tx / Distributed Tx / XA Tx

⇒ Uses multiple resources / DBs in a single transaction boundary. Allows to perform commit or rollback on multiple DBs / resources in a single shot.

⇒ To work with Global Tx we need XA jdbc drivers, XA datasources, and XA DBs/ws.

⇒ Oracle all versions, mysql 5.x+ are XA DataSources Bases..

⇒ For managing distributed Tx we need "Jta Transaction Manager" and it expects
a) transactionManager b) userTransaction
We can supply both values from Application server through jndi lookups or we can use Third party api like "atomikos" to supply the same.

⇒ use case : transfermoney operation b/w two accounts of two different banks (HDFC, ICICI)

```
<bean id="atomikosTxManager" class="com.atomikos.icatch.jta.UserTransactionManager"/>
<bean id="atomikosUT" class="com.atomikos.icatch.jta.UserTransactionImpl"/>
<bean id="dtxmgr" class="org.sf.transaction.jta.JtaTransactionManager">
  <property name="transactionManager" ref="atomikosTxManager"/>
  <property name="userTransaction" ref="atomikosUT"/>
</bean>
```

Example

TxMgmt Proj 4 (Anno-Distributed Tx)

```
|->src
   |-> com.nt.cfgs
         |-> applicationContext.xml
   |-> com.nt.service
         |-> AccountSevice.java
   |-> com.nt.dao
         |-> WithdrawDAO.java
         |-> DepositDAO.java
         |-> WithdrawDAOImpl.java
         |-> DepositDAOImpl.java
|->test-> clientApp.java
```

atomikos-transaction-jta.jar
atomikos-util-3.7.0.jar
com.mysql.jdbc_5.1.5.jar
jta.jar
ojdbc14.jar
transactions-3.7.0.jar
transactions-jdbc-3.7.0.jar

jar files : SpringJDBC LIB + AOP LIB + aspectjrt.jar + aspectjweaver.jar + atomikos-transaction-api.jar

DB tables in oracle

tx-account (table)

accno	holder	balance
101	raja	9000

DB tables in mysql 5.x (logical db : ntsept)

tx-account (table)

accno	holder	balance
102	ravi	8000

⇒ While working with Distributed Tx, the UserTransaction object represents transaction service and TransactionManager object represents the transactionManager that performs commit (or) rollback by using transaction service.

⇒ JtaTransactionManager can use either server supplied (or) third party api supplied transactionMgr, usertransaction object to perform Tx management.

8/9/15

Different models of Webapplication development

M → Model
V → view
C → controller

Model1

Here either jsp or servlet will be taken as main web resource prgs having multiple logics.. Mainly they prefer using jsps.

⇒ Since every servlet/jsp contains multiple logics there is no seperation btw logics. So this architecture is not suitable for large scale web appln.

Model2 (MVC)

MVC1

MVC2

⇒ In Model2 we take multiple technologies in multiple layers to develop various layers. This gives clean seperation bit the logics.

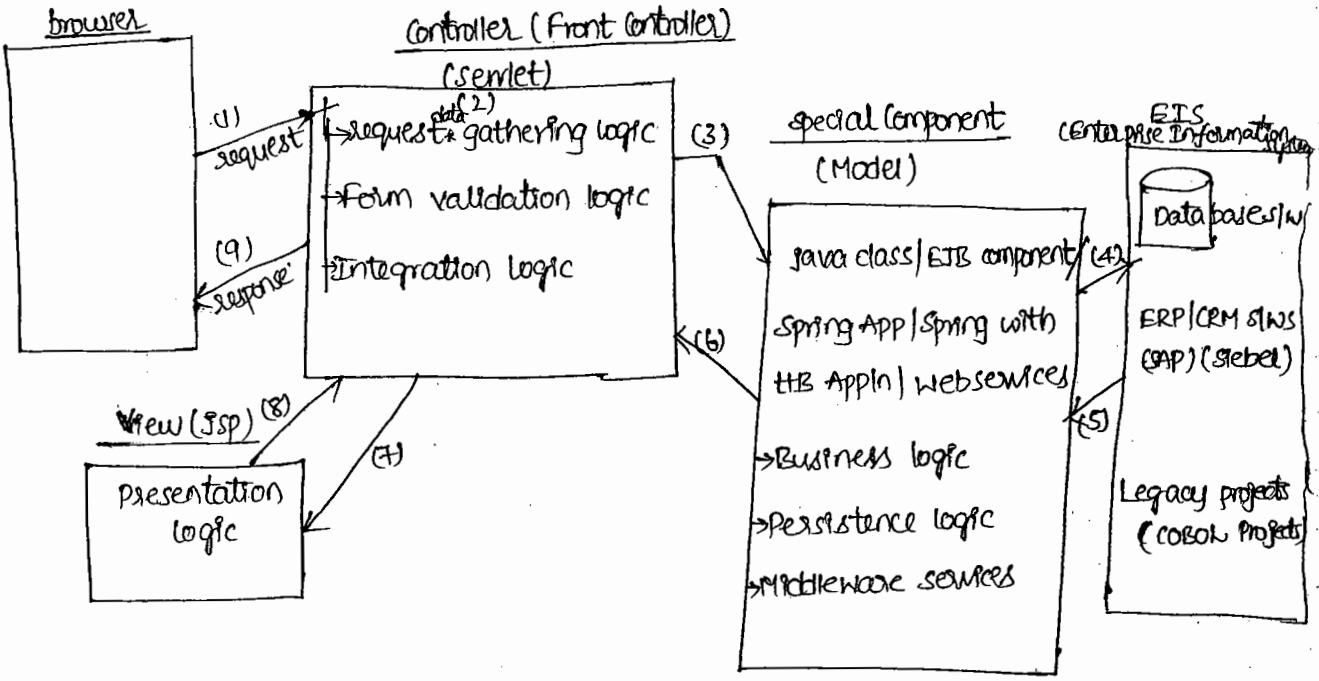
⇒ In MVC1 architecture we take single resource (servlet/jsp) having view, controller logics and we take seperate resources for model layer logics.

⇒ In MVC2 architecture we take seperate resources for view, seperate resources for model and seperate resources for controller.

M → Model → Data + Business logic + persistence logic
V → view → presentation logic
C → controller → Integration logic / controller logic.

controller maintains logics to control all the activities of web application like takes the request, passes the request to appropriate Model component, gathers the results from model and passes the results to view.

MVC2 Architecture



⇒ Multiple Back end SWs together is called EIS. (Enterprise Information System).

⇒ The Project that is developed by using old technologies or old versions of existing technologies is called Legacy project. e.g: COBOL and Java 1.0 projects.

Advantages of MVC2

⇒ Because of multiple layers we get clean seperation b/w logics.

⇒ The modification done in one layer logics does not effect on other layers logics.

⇒ Maintainence and enhancements of the logics becomes quite easy.

⇒ Parallel development is possible, so gives good productivity.

⇒ While using EJB, Spring in the model layer we can take advantage of built-in middle ware services.

Disadvantages of MVC2

⇒ knowledge on more technologies is required

⇒ for parallel development more programmers are required.

⇒ Every web application contains some commons logics to handle along with application specific logics.

a) Request Graphing

→ Reading data from the form component of form page and placing the data in a java class object

b) Form submission / Request submission.

⇒ We get request from different places of web pages like through hyperlinks, through submit buttons (or) through other components.

→ These request may come with (or) with out data, we should handle all request properly.

c) Form validation / Form Handling

→ verifying the pattern, format of form data and displaying same form page with errors having old values in form components.

⇒ If we develop web application by JEE technologies directly we need to write the above common logics in multiple servlet programs which leads to Boiler plate code problem.

⇒ If we develop MVC2 architecture web application by using JEE technologies directly (servlet/jsp) we need to develop all the logics of all layers manually, More over we need to write both common and application specific logics.

3/9/15

MVC2 Architecture / Design pattern rules

⇒ Every layer is designed to have specific logics, so place only those logics and do not place any additional logics.

⇒ All the operations of the application must take place under the control or monitoring of controller (servlet).

⇒ There can be multiple resources in view layer, multiple resources in model layer but there must be only one servlet acting as controller. This must be taken as Front Controller.

⇒ The view and model layer components must not interact with each other directly. They must interact through controller.

* If we develop MVC2 architecture based webapplication manually all the logics of all layers (including common, application specific logics) must be taken care by programmer.

⇒ If we develop MVC2 web application by using web framework then the common logics will be taken care by the framework by giving one built-in servlet as Front Controller so the programmer just need to develop only application specific logic.

⇒ The common logics are like request graphing, form/request submission, form handling.

⇒ The web frameworks are

a) Struts → from Apache (4)
b) JSF → from SunMS (oracle corp) (2)
c) Web Work → from open symphony (5)
d) ADF → from oracle corp (3)
e) Spring MVC → from Interface21 (1)

⇒ The special web component of web application that acts as entry point for all requests and applies common system services for all requests like request graphing, request services, form handling is called front controller.

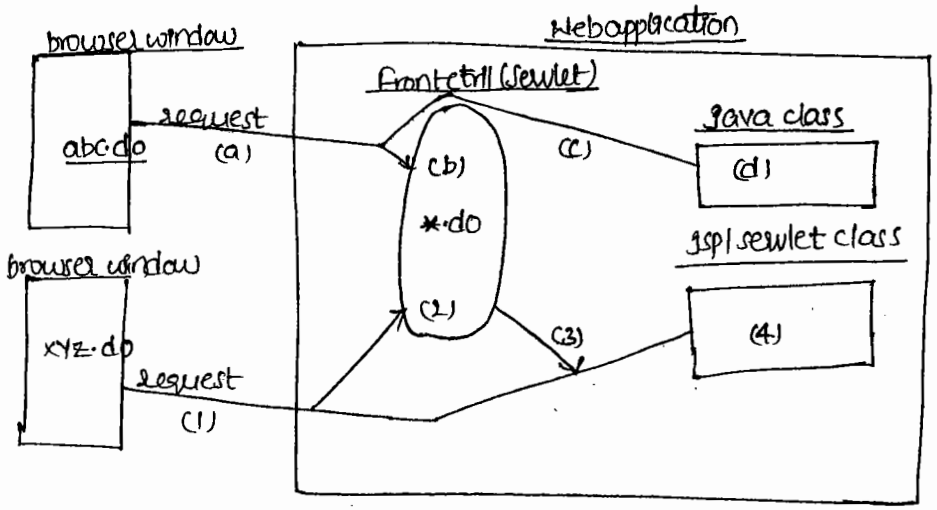
⇒ We can take either servlet or JSP as front controller. servlet is always recommanded.

⇒ We take front controller trapping and taking multiple requests coming from client, we must configure them either with directory match or extension match url pattern.

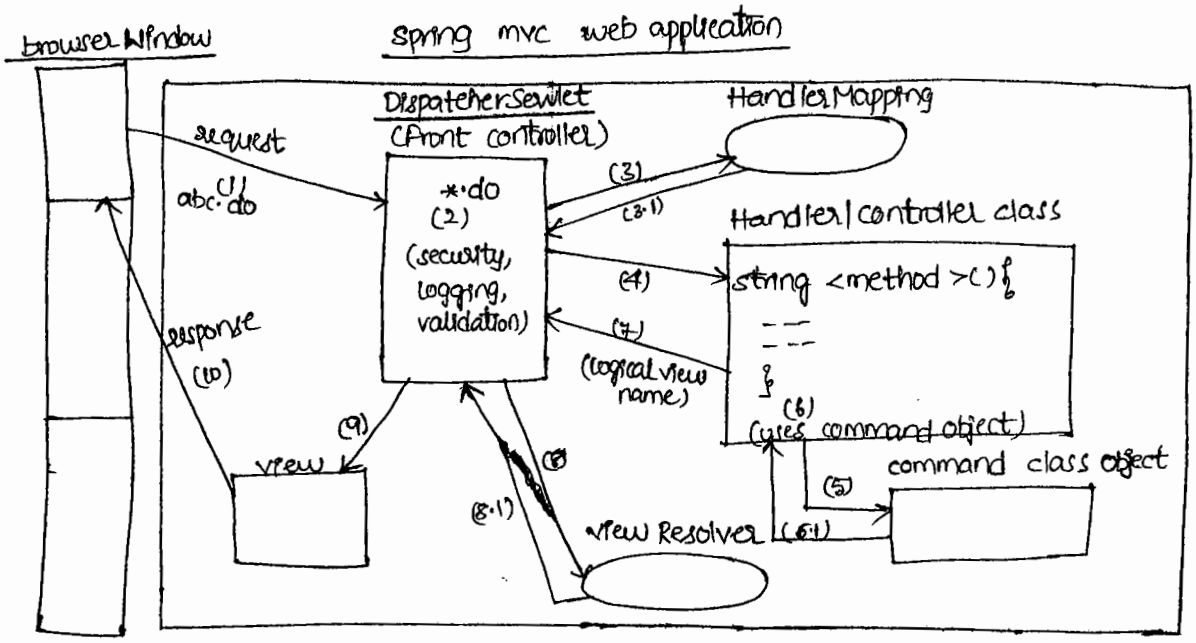
→ *.do, *.htm, *.http... are extension match url pattern.

→ /*, /x/*, /x/y/*,.. are Directory match url pattern.

⇒ In struts, ActionServlet is built-in front controller servlet. In spring MVC, DispatcherServlet is built-in front controller.



Spring MVC web application Flow



⇒(1) Browser gives request to webapplication.

(2) As front controller, dispatcher servlet traps and takes the request and applies the common system services like security and etc..,

(3) DispServlet uses HandlerMapping component to decide the handler class to utilize based on the incoming request URI.

(4) DispServlet passes the controller to handler class by calling method.

(5) Handler class internally writes the received form data to command object.

(6) Handler class process the request and generates the output. If needed it also uses command object to work with form data.

(7) Handler class returns logical view name back to dispatcher servlet.

(8) Disp servlet uses view resolver to get view object having view layer technology and resource name.

(9) Disp servlet uses the view object to pass the controller to view resource,

(10) The view resource format the results and sends the output (response) to browser.

4/9/15

Spring MVC features

a) Allows to use various technologies in the view layer like html, JSP, velocity, freemaker, xslt and etc..

b) Allows to design command class (whose object holds form data) as POJO class (need not to extend or implement spring api class/ interfaces).

c) Allow to take any type properties in command class to hold form data.

d) Supports Internationalization (I18n). [Making our appln working for different locales]

e) Allows to configure themes for rendering better presentation.

f) Allows to develop business logics, form validation logics as reusable logic.

g) Provides clean seperation b/w logics/components by involving Handler mappings, views, viewResolvers, command classes, Handler/controller classes and etc...

h) Allows to integrate with other modules of spring framework.

⇒ Spring MVC is designed around DispatcherServlet because various operations on the request will be done by DispatcherServlet by using different components to process the request and to deliver the response.

⇒ Even though it is predefined servlet its configuration in web.xml is mandatory. we configure this servlet either with extension match or directory match url pattern to make that servlet as front controller to take multiple request.

⇒ This servlet internally activates IOC container of type web application context by taking

<DispatcherServlet logicalname>-servlet.xml or specified filename (as namespace init param value) as springBean cfg file.

⇒ SpringBean cfg file contains Handler mappings, view resolvers, views, Handler/controller classes object.

web.xml

```
<web-app>
  <servlet>
    <servlet-name> dispatcher </servlet-name>   ← logical name
    <servlet-class> org.sf.web.servlet.DispatcherServlet </servlet-class>
                                                  [ds-dispatcherservlet]
    [<init-param>
      <param-name> namespace </param-name>
      <param-value> applicationContext </param-value>
    </init-param>]
    <load-on-startup> 2 </load-on-startup>
  </servlet>
```

* If this namespace init param is not cfg the IOC container created by ds will use dispatcher-servlet.xml as spring bean cfg file.

* If it is configured, the IOC container take application context as spring bean cfg file.

```
<servlet-mapping>
    <servlet-name> dispatcher </servlet-name>
    <url-pattern> *.htm </url-pattern>
</servlet-mapping>

</web-app>
```

⇒ It is recomanded to place jsp files in web appln inside WEB-INF folder or) in its sub-folders.

The reasons are

1) We can hide the JSP technology that is used in web appln from end users.

2) If jsp page is displaying request attributes given by servlet, the direct request to jsp will display ugly values. We can avoid this ugly messages by not allowing direct access to jsp.

For this, place jsp inside the WEB-INF.

3) If session is not logged out properly hackers can access previous session by appending session id to the request url of jsp collected from browser history. This can be avoided by keeping jsp's inside the web-inf.

⇒ In order to develop our controller/Handler class we must take a class implementing org.sf.web.servlet.mvc.Controller Interface and should provide implementation for handleRequest() method having logic to process the request and to return ModelAndView obj having logical view name.

⇒ for example

```
public class MyController implements Controller {

    public ModelAndView handleRequest(HttpServletRequest req, HttpServletResponse res)
    {
        ---
        return ModelAndView("home");
                              ↳ logical view name
    }
}
```

5/9/15

⇒ Every controller/Handler class must be configured in spring bean configuration file having bean id. The bean id must be mapped with incoming request url by using one of another handler mapping.

⇒ HandlerMapping is responsible to map incoming request url through Handler classes (controller class). Every HandlerMapping is a class that implements org.sf.web.sewlet.HandlerMapping Interface. The most regularly used HandlerMapping class is SimpleURLHandlerMapping.

Sample code

```
<!-- Controller class -->
<bean id="wc" class="com.nt.controller.WelcomeController"/>

<!-- HandlerMapping -->

<bean id="surl" class="org.sf.web.sewlet.handler.SimpleUrlHandlerMapping">
    <property name="mappings">
        <props>
        <prop key="home.htm">wc</prop>
        </props>
    </property>
</bean>
```

First spring MVC web application that displays "welcome.jsp" of WEB-INF/pages folder upon receiving the request

```
MVC (FirstWeb)
 |--> JavaResources
 |      --> src
 |            --> com.nt.controller
 |                  --> WelcomeController.java
 |
 |--> WebContent
        --> WEB-INF
              --> pages
              |     --> welcome.jsp
              |
              --> web.xml
              --> dispatcher-sewlet.xml
```

Build Path

Jar files : SPIOC LIB + spring-web-4.1.6.RELEASE.jar + spring-webmvc-4.1.6.RELEASE.jar

(perform Deployment Assembly)

Web.xml (b)

<refer previous class (Donot configure namespace init-param)>

WelcomeController.java

(f)

```
public class WelcomeController implements Controller {

    @Override
    public ModelAndView handleRequest(HttpServletRequest req, HttpServletResponse res)
                                                    (g)
            throws Exception {
                                    request attribute name
        return new ModelAndView("welcome", "msg", "WelcomeTo Spring MVC");
                    (h)          logicalViewName          request attribute value
    }

}
```

dispatcher-servlet.xml (c)

```
<bean id="wc" class="com.nt.controller.WelcomeController" />   (e)
<!--HandlerMapping-->
<bean id="surl" class="org.sf.web.servlet.handler.SimpleUrlHandlerMapping">
    <property name="mappings">
      <props>
        <prop key="home.htm">wc</prop>   (d)
      </props>
    </property>
</bean>

<!--View Resolver-->
(i) <bean id="vr" class="org.sf.web.servlet.view.InternalResourceViewResolver">
    <property name="prefix" value="/WEB-INF/pages/" />
                                                    (j)
    <property name="suffix" value=".jsp" />
                                    (k)
</bean>
```

Welcome.jsp

```
<h1><center> HOME PAGE </center>
    ${requestScope.msg}
```

request url: http://localhost:3030/MVC(FirstWeb)/home.htm (a)

View Resolver

⇒ This view Resolver takes the logical view name given by controller class and resolves physical view name by adding prefixes, suffixs and returns view interface implementation class object back to dispatcher servlet having physical view name.

⇒ DispatcherServlet calls render() method given view Implementation class object and this render method passes the control to a physical view resource.

⇒ Every view Resolver is a class that implements org.sf.web.servlet.ViewResolver interface.

⇒ Spring has given multiple built-in view resolver classes in that InternalResourceViewResolver is useful to return view implementation class object having the servlet/jsp programs placed in WEB-INF folder of the webapplication (internal resources).

⇒ Spring MVC2 Appln to that launches home page and process the request based on the hyperlink clicked in the home page.

Note In spring MVC only welcome pages will be placed outside the WEB-INF folder/ pages and remains jsps are recommanded to place inside WEB-INF folder for security reasons. welcome page executes automatically for every request given to web-app and gives implicit request by using <jsp: fowoard> tag response.sendRedirect() method to launch other pages that are there inside WEB-INF folder through controller/ handler class.

(a)
index.jsp (welcome page)

```
<jsp:fowoard page="home.htm"/>
```
(b)

Dispatcher-servlet

(Front Controller)
(c)
*.htm
(j)

SimpleUrlHandlerMapping
(d)
⊢→home.htm ↔ hhc
⊢→ wish.htm ↔ wc
(k)

home.jsp

```
<a href="wish.htm">
    wish Msg </a>
```
(i)

(h) (n)
view object
↑
(g)

InternalResourceViewResolver
⊢→prefix : /WEB-INF/pages/
⊢→suffix : .jsp

(e)
```
public class VrewHomeController
    extends AbstractController
{
  public MAV handleRequest
          Internal (req, res)
  {
  return new MAV("home");
  }
}
```
(f)

result.jsp (P)

```
wish msg :{$ (wmsg)}
```

O/P Wishes

(WC) (L)

```
public class WishController extends
        AbstractController {

public MAV handleRequestInternal(req, res){

  ---//b.logic to generate wishmsg

return new MAV ("result", "wishmsg", "--");
                  (M)
 }
}
```

"result" → logical View Name; "wishmsg" → request attribute Name; "--" → request attribute value

MVC Second Web

```
|--> java resources
    |--> src
        |--> com.nt.controller
            |--> ViewHomeController.java
            |--> WishController.java

|--> webcontent
    |--> index.jsp
    |--> WEB-INF
        |--> pages
            |--> home.jsp, result.jsp
        |--> web.xml
        |--> dispatcher-servlet.xml
```

refer page No:2 of MVC

Jars: Same as first App.

⇒It is always recommanded to develop an controller class by extending from pre-defined xxx controller classes that means it is not recommanded to develop same class by implementing Controller Interface directly.

For the above sample code based ex Appln (Near by) refer App1 of page No:1 of MVC.

7/9/15

Controllers | Handlers

⇒ The java class that implements org.sf.web.sowlet.mvc.Controller is called controller | Handlers. It is mainly useful to render logical view name and data that is required to display in view layer resources.

⇒ Placing business logic | request processing logic in controller classes is not recomanded process. It is recomanded to place such logic in sewice classes and comunicate with them from controller classes.

⇒ Spring api supplies lots of built-in controller classess implementing controller interface. These classes are given to handles the requests coming from form pages, hyperlinks and other form components. The implementation controller classes are

a) Abstract Controller
b) Parameterizable View Controller
c) UrlFileName View controller
d) Abstract Command Controller
e) SimpleFormController
f) Controller
g) MultiActionController
h) AbstractWizardFormController

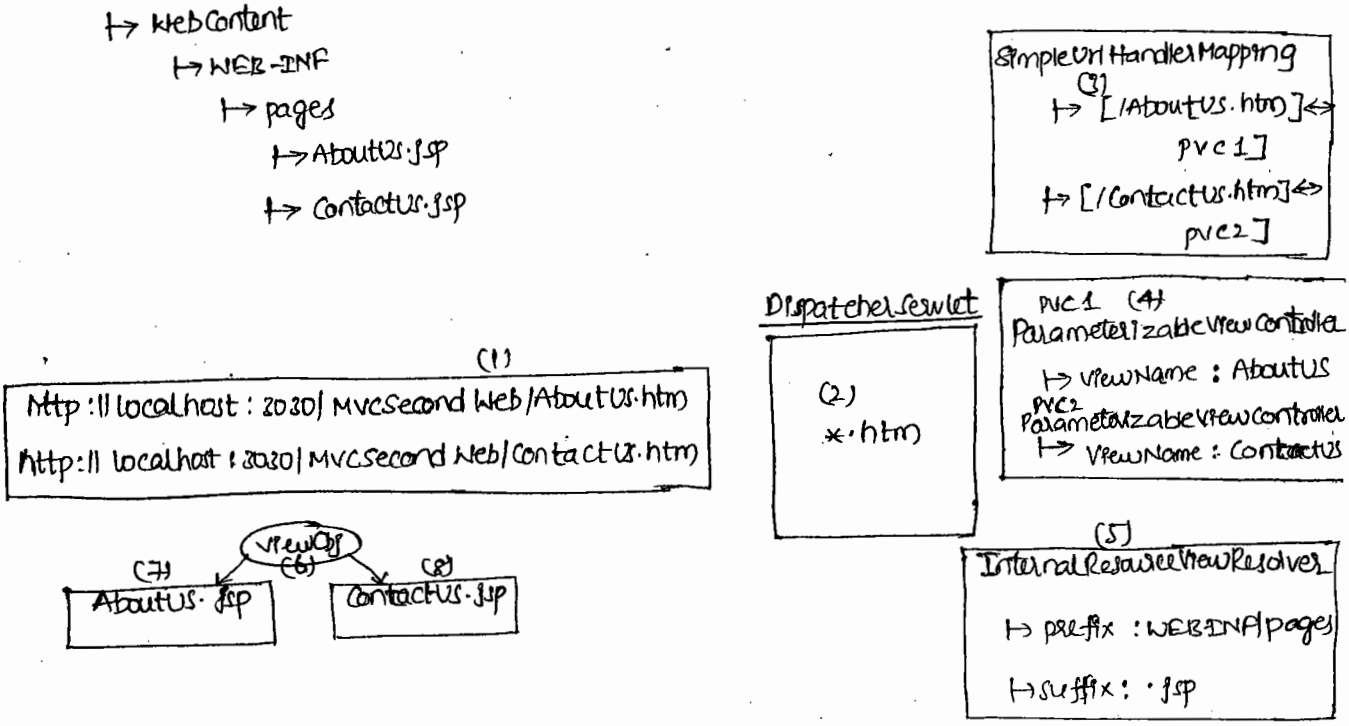
i) Parameterizable View Controller

⇒ In order to display webpage with processing request then instead of taking seperate user-defined controller class we can configure this class by specifying the destination view name "viewName" property value. This is useful to display webpage for hyperlink generated requests without having any request processing.

eg: working with "AboutUs", "contactUs", hyperlinks.

⇒ For multiple requests of kind we need to configure this class for multiple times.

MVCSecondWeb
- WebContent
  - WEB-INF
    - pages
      - AboutUs.jsp
      - ContactUs.jsp

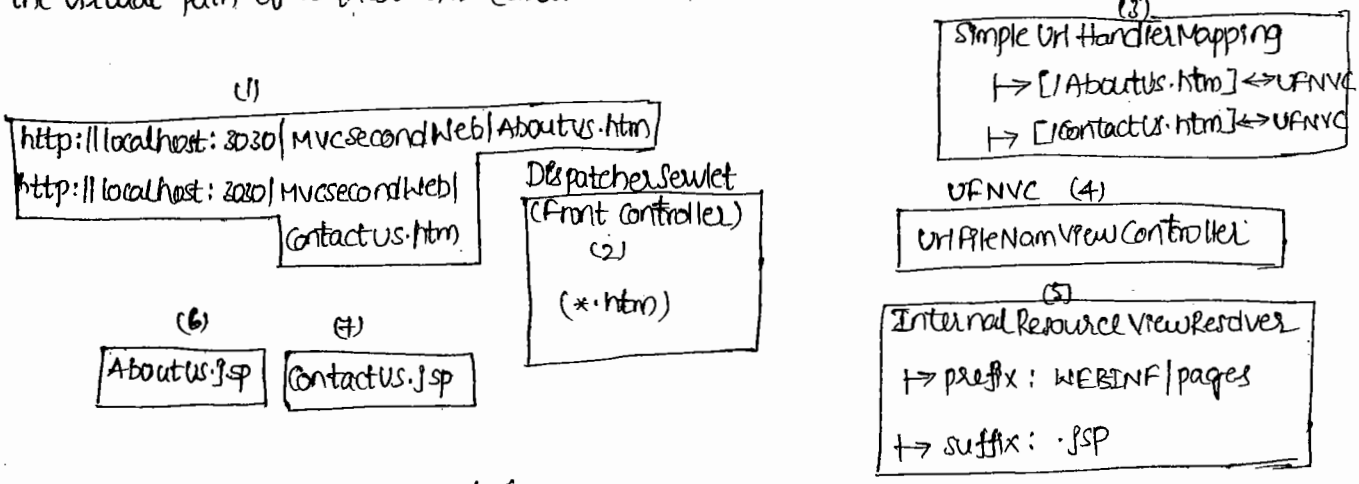


(2) UrlFileNameViewController

⇒ If we want to return the virtual path of (like aboutus.htm) of incoming request URL as logical view name to render view/webpages with out having any request processing then we can use this controller.

⇒ This is alternate to using multiple ParameterizableViewController classes configuration to render multiple view pages/views with out having any request processing for multiple hyperlinks generated requests like aboutus, contactus and etc.

⇒ While configuring this class no need of specifying any property because it takes the virtual path of request URL (aboutus.htm) and return it as logical view name automatically



Note: If request is there with out form data (from hyperlinks) and no request processing is required but displaying views is required then go for ParameterizableViewController (or) UrlFileNameViewController

⇒ In real time every spring appln will be designed to have 2 IOC containers as parent, child container because it is recomanded to maintain presentation tier components like Handler mappings, View Resolvers, Controller classes in 1 container (WebApplication-Context given by DispatcherServlet) and Business tier components like service class, DAO class, AOP Aspect classes in another container (the applicationcontext IOC container given by ContextLoaderListener).

⇒ ContextLoaderListener is a spring supplied predefined servletContextListener which activates applicationContextListener when servletContextListener object is created either during server startup or during the deployment of webapplication. By taking param-name as contextConfigLocation and context param value as spring bean configuration file.

Sample code for web.xml

```
<web-app>
 <servlet>
     <servlet-name> dispatcher </servlet-name>
     <servlet-class> org.sf.web.servlet.DispatcherServlet </servlet-class>
    <load-on-startup> 2 </load-on-startup>
 </servlet>
 <servlet-mapping>
  <servlet-name  >dispatcher </servlet-name>
 <url-pattern>*.htm</url-pattern>
 </servlet>
 <listener>
 <listner-class> org.sf.web.context.ContextLoaderListener </listener-class>
 </listener>
  <context-param>
     <param-name> contextConfigLocation </param-name>
     <param-value> /WEB-INF/applicationContext.xml </param-value>
  </context-param>
</web-app>
```

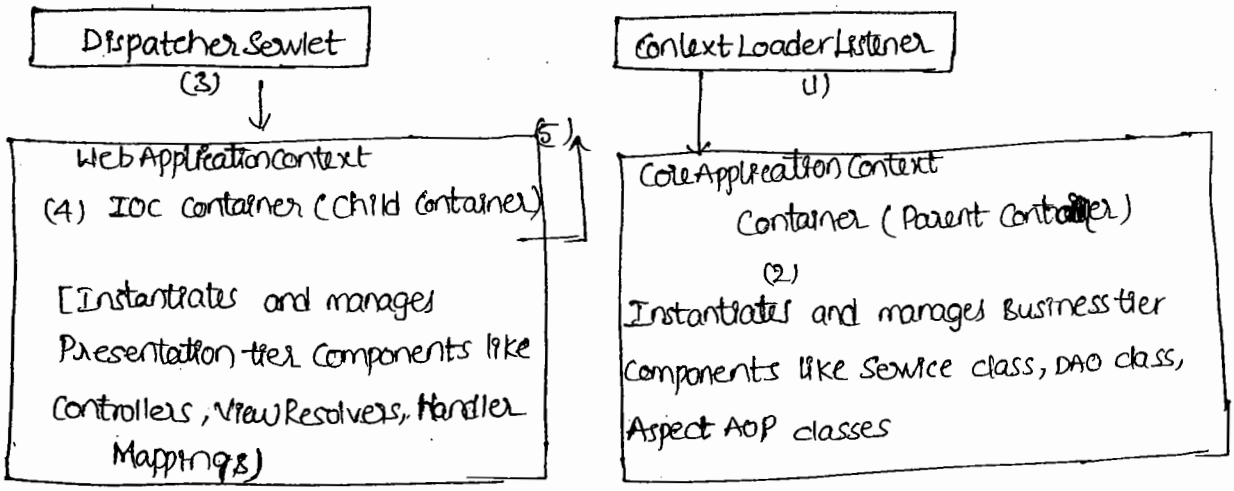
⇒ If Name Space init parameter is not configured for DispatcherServlet then the activated web appln context container takes <servlet name>-servlet.xml as spring bean cfg xml.

eg: (dispatcher-servlet.xml)

⇒ Generally ServletListener object will be created first compared to the servlet objects on which load-on-startup is enabled. But some servers creates servlet object first before listener object if load-on-startup priority value is 1.

⇒ We want to see dispatcher servlet web application context container as child container to ContextLoader's listener application context container in all situation and all servers for this it is recomanded to enable load-on-startup on DispatcherServlet with priority value other than one (1).

8/9/15



; Since Controller classes are managed by beans in child container so they can be injected with service class objects | beans managed by Parent Container.

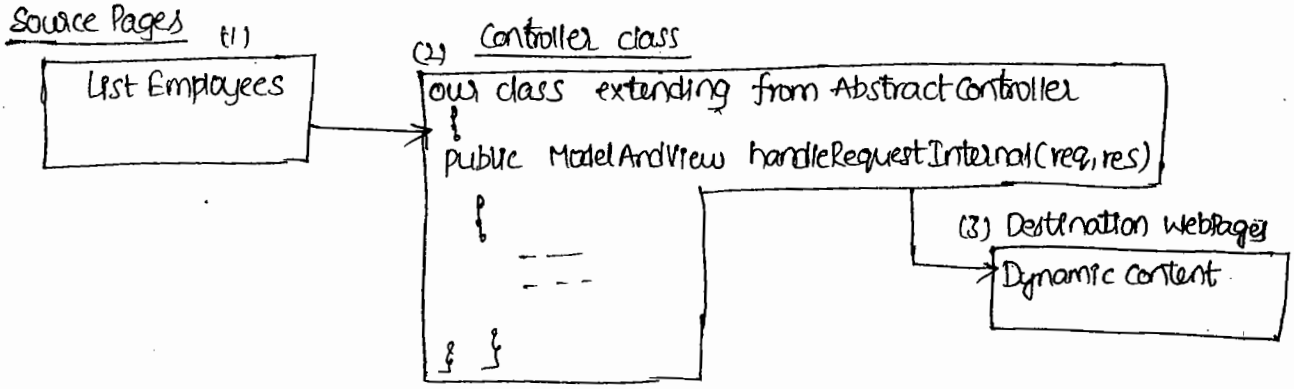
3) Abstract Controller

⇒ Take this class to develop controller class when there is no form submission in request but request processing required to display dynamic webpage as output. Generally for hyperlink generated requests to get dynamic content having request processing we go for AbstractController class.

Usecase 1) getting current user profile in facebook when "viewProfile" link is clicked,

2) Getting all the jobs that are posted when "current trends" link is clicked.

3) Getting all employees details when "list Employees" hyperlink is clicked.

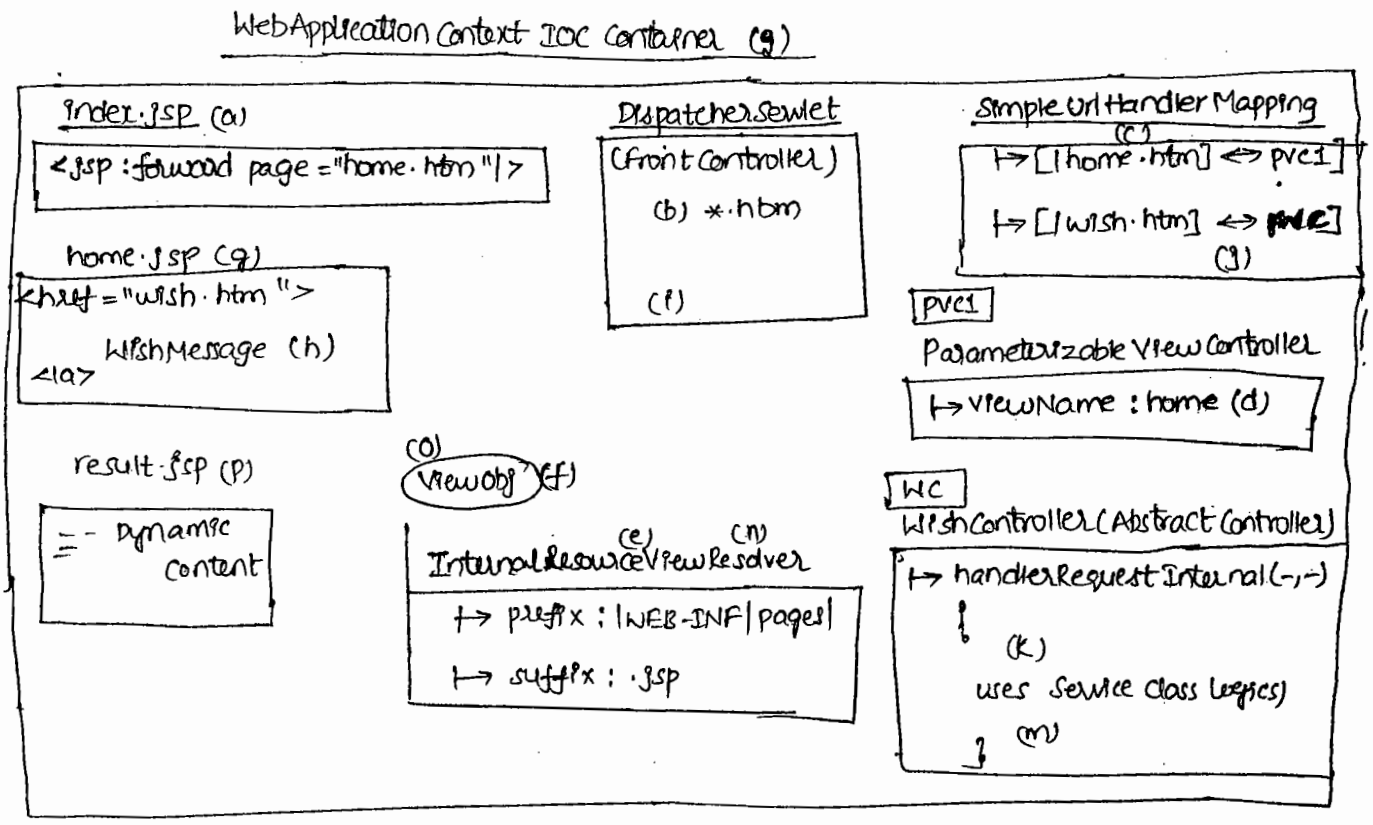


a) Here no form submission, so there will not be any request graphing, form handling

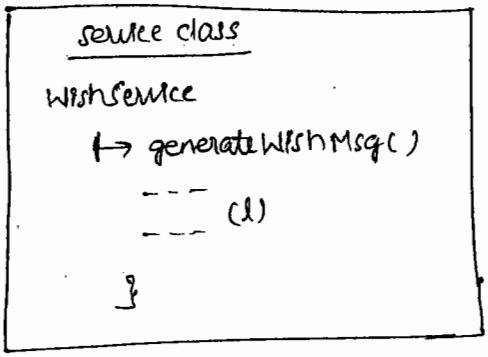
b) Since there is no form submission the handleRequestInternal(-,-) does not take a command object as argument.

c) handleRequestInternal(-,-) returns ModelAndView object having logical view name and Model Data that has to be displayed as argument. dynamic content.

Note Writing business logic in controller class is not a good pratice, always place this logic in Service class and communicate from controller class.



Application Context IOC container (1)



MVC WishApp3.

- JavaResources
  - src
    - com.nt.controller
      - WishController.java
    - com.nt.service
      - WishService.java
- Webcontent
  - index.jsp
  - WEB-INF
    - pages
      - home.jsp
      - result.jsp
    - web.xml
    - dispatcher-servlet.xml
    - applicationContext.xml

9/9/15

(service-beans.xml)

⇒ In real time, beans of service layer will be configured in seperate xml file similarly the beans of persistence layer will be configured in seperate xml file and all these xml's will be imported in applicationContext.xml file.

persistence-beans.xml

MVCEmpslistApp4 (AbstractController) Flow;

WebApplicationContext Container (child container)

index.jsp (1)

```
<jsp:forward page="home.htm" />
```

DispatcherServlet

```
*.htm
(2)
(9)
```

SimpleUrlHandlerMapping

```
|-> [/home.htm <-> pvc1] (3)
    |-> [/listEmps.htm <-> lec] (10)
```

pvc1

ParameterizableViewController (4)

```
|-> viewName : home
```

home.jsp (7)

```
<a href="listEmps.htm">
  All Emp Details </a> (8)
```

(19) (6) ViewObject

(5) (18) InternalResourceViewResolver

```
|-> prefix : /WEB-INF/pages
|-> suffix : .jsp
```

lec (11)

ListEmployeeController

```
|-> handleRequestInternal
        (req, res)
List ListEmpDTO = service.getAllEmps(); (12)
return new MAV("listEmps",
                "empList",
        (17)    "listEmpDTO");
}
```

listEmps.jsp (20)

```
-- //Displays Empdetails
   as html table content
```

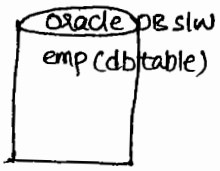
ApplicationContext Container (parent container)

ListEmployeeSevice

```
|-> getAllEmps() (13) {
    List listEmpBO = dao.retrieveEmp(); (14)
    //store ListEmpBO obj in ListEmpDTO objs
    ---
    return listEmpDTO;
    (16)
}
```

ListEmpsDAO {

```
|-> retrieveEmps() (15) {
    ---
    ---
    return ListEmpBO objects;
    }
}
```



MVCEmpsListApp4 (AbstractController)

- JavaResources
  - src
    - com.nt.service
      - ListEmpService.java
    - com.nt.dao
      - ListEmpDAO.java
    - com.nt.bo
      - EmpBO.java
    - com.nt.dto
      - EmpDTO.java
    - com.nt.controller
      - ListEmpController.java
- webcontent
  - index.jsp
  - WEB-INF
    - pages
      - home.jsp, listEmps.jsp
    - web.xml
    - dispatcher-servlet.xml
    - service-beans.xml
    - persistence-beans.xml
    - applicationContext.xml

jars: same jars
+
jstl.jar, standard.jar
+
org.sf.jdbc <ver>.jar
+
org.sf.tx <version>.jar
+
ojdbc6.jar

11/9/15

Command class

⇒ The java bean class that will be instantiated by spring MVC to store the form data of form page is called 'command class'. The command class property names and form page component names must match. This avoids programmers burden of calling request.getParameter() method and writing form data to command java object.

⇒ Command class properties can be of any type to hold different types of values given by form page.

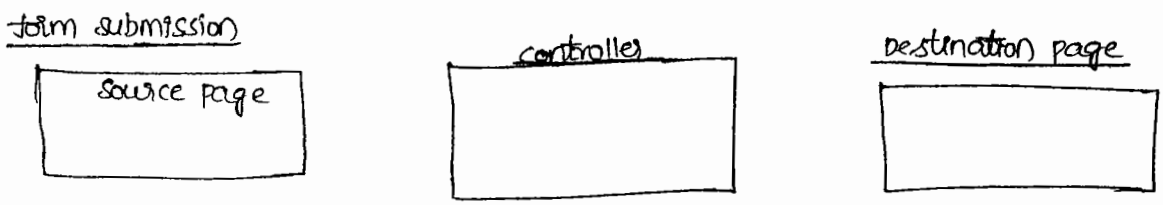
AbstractCommand controller

⇒ This controller is capable of reading form data of formpage and writes that data to command class ie, it takes care of request graphing | wrapping.

⇒ use this controller when source page is form submission with no server side form validations, optional client side form validations and no form handling (No need of displaying same page with old values) and request wrapping required, (destination page should be static or dynamic page that should come after request processing.

⇒ When we develop a controller class extending from AbstractCommandController we must implement handle() method.

```
public ModelAndView handle(HttpServletRequest req, HttpServletResponse res, Object cmd,
                           BindException be)
```



Source page

⇒ Form with submit button (or) java script based form submission having optional client side form validation

⇒ Since form is there request wrapping | graphing is required.

⇒ No form validation.

Controller

⇒ Receives form data and stores in command class object.

⇒ Process the request by taking support of service, DAO classes.

⇒ Returns ModelAndView object having logical view name.

Destination Page

⇒ static / Dynamic content that comes after request processing.

Usecase

Searching employee details based given no details or 1|2|3|— details in a form.

⇒ While configuring Controller class whose type is AbstractCommandController we must specify property "command class" having fully qualified command class name as value.

Web Application Context

index.jsp (a)

```
<jsp:forward
    page = "search.htm" />
```

DispatcherServlet

(b) *.htm (j)

SimpleUrlHandlerMapping

(c) ↦ [search.htm ⇔ pvc1]

↦ [searchEmps.htm ⇔ esc] (k)

(d) pvc1

ParameterizableViewController

↦ viewName : search

(e) (t) ViewObject (f)

search.jsp (g)

(i)

```
<form action="searchEmp.htm">
No : <input type = "text" name = "no" />
Name : <input type = "text" name = "name" />
Design : <input type = "text" name = "degn" />
Salary : <input type = " name = "salary" />
<input type = "submit" value = " search" />
</form>
```

(h)

(s)

InternalResourceViewResolver

↦ prefix : /WEB-INF/pages

↦ suffix : .jsp

Command class

```
public class SearchCommand {
private int no, salary;
private String name, desg;
//setters and getters
--
}
```

esc

EmpSearchController (AbstractCommandController)

↦ commandClass : <pkg>.SearchCommand

↦ handle (req, res (l), cmd, BE) {

```
//convert cmd obj to DTO obj.
ListDTO = service.process (dto);   (m)
return MAV ("ListEmps", "_", "_");   (r)
}
```

ListEmps.jsp (v)

display search results in the form of html table.

Application Context

EmpSearchSeviceImpl

```
(1)
|→ process(dto){
   convert dto to BO object
   //use DAO class
   dao.search(BO);   (2)
ListRBO = dao.search(BO);
   //convert ListRBO to ListRDTO
   ---
   return ListRDTO;   (4)
}
```

EmpsearchDAOImpl

```
(3)
|→ search(BO){
   --
   ---
return ListRBO>;
}
```

MVCEmpSearchApp5 (AbstractCommandController)

```
|→ Java Resources                    Jars: same as MVCEmpListApp4.
   |→ src
      |→ com.nt.controller
         |→ EmpSearchController.java
      |→ com.nt.service
         |→ EmpSearchService.java
         |→ EmpSearchServiceImpl.java
      |→ com.nt.dao
         |→ EmpSearchDAO.java
         |→ EmpSearchDAOImpl.java
      |→ com.nt.bo
         |→ SearchBO.java, SearchResultBO.java
      |→ com.nt.command
         |→ SearchCommand.java
      |→ com.nt.dto
         |→ SearchDTO.java, SearchResultDTO.java
|→ WebContent
   |→ index.jsp
   |→ WEB-INF
      |→ pages
         |→ search.jsp, ListEmps.jsp
      |→ web.xml, dispatcher-servlet.xml, service-beans.xml, persistence-beans.xml, aop-beans.xml, applicationContext.xml
```

12/9/15

⇒ The movement we enable Tx the underlying data base software will be applied with read committed isolation level that means it allows us to read only committed data from database.

13/9/15

⇒ For every request given our AbstractCommandController class, one object of command class will be created for request wrapping/graphing.

⇒ Dispatcher Sewlet always get our controller class object from web application context container using ctx.getBean(-) and calls handleRequest(req,res) on that object. If not haavailable the super class handleRequest(-,-) executes..

DispatchSewlet

(a)
controller.handleRequest(req,res);
(our class object)
(h)

```
public class AbstractCommandController implements
                                    Controller{
public MAV (b) handleRequest(req,res){
→ creates Command class object writes the
  received data to that object
→ (c) creates an Empty BindException class object
→ (d) calls handle(-,-,-,-) with req,res, command
  class object, BindException object
→ (g) returns MAV object given by handle(-,-,-,-) call
}
public abstract MAV handle(req,res,Object cmd,
                           BindException);
}
public class EmpSearchController extends AbstractCommand
                                    Controller{
public MAV (e) handle(req,res, object cmd, BindException be)
{
---
---
return new MAV(-,-,-); (f)
}
}
```

⇒Dispatcher servlet does not create command class object the handleRequest(-,-) of super class to our controller class like AbstractCommandController/SimpleFormController/--- will create this command class object.

5) SimpleFormController

⇒It is multipurpose controller / It can be used for form submission, form validation, request wrapping, solving double posting and etc...

⇒If our source page is form page having form validation, form handling and etc.., and looking to get Destination page either as static / dynamic page having request processing then use simpleFormController.

⇒ SimpleFormController

properties

- commandClass
- commandName → Attribute name when command obj is stored as request/session attribute
- formView : form page name
- successView : result page name
- sessionForm and etc...
  - true → keeps command object in session scope
  - false → keeps command object in request scope

Methods

- public MAV onSubmit(Object cmd)
- public MAV onSubmit(object cmd, BindException be)
- public MAV onSubmit(req, res, Object cmd, BindException be)

Source page	SimpleFormController	Destination page
→ form page/ form submission → request wrapping → form validation → form handling	→ commandClass → commandName → formView → successView MAV onSubmit()	→ static/Dynamic web page

⇒The SimpleFormController displays form page based on "formView" value if the request is initial phase request (request method: GET) i.e, there is no need of taking seperate ParameterizableViewController for lauching form page.

⇒The same controller becomes ready to process the request if the request is PostBack request (request method : Post)

1) DispatcherServlet calls handleRequest(req,res) on our SimpleFormController class obj
2) Since not available, it will call super class (SimpleFormController) handleRequest(-,-) method executes
3) The handleRequest(-,-) checks weather request is initialphase request or post back request

* Initial Phase request (if req.getMethod.equals("GET")) {
  a) Creates an empty Command class object
  b) keeps Command class object either in session/request scope
  c) gather formView property value and returns it as logical view name to DispatcherServlet
  d) gather ~~form~~ ViewResolver takes logical view name and gives View object to render form page to browser. This form page can display the initial values by collecting from command object.

* Post back Request (if req.getMethod.equals("POST")) {
  a) Creates an ~~empty~~ Command class object and writes form data to it (request wrapping)
  b) creates an empty BindException object
  c) calls onSubmit() having req, res, command object, BindException object
  d) onSubmit() our Controller class will process the request and returns MAV object having logical view name.
  e) ViewResolver takes logical view name and gives View object to render result page.

Note While working with SimpleFormController our form page request method must be "POST".

⇒In form handling we need to display form page with old values when form validation errors are there in this process we need to write lot of additional code using javascript/jsp use bean tags if we work with traditional html tag. If we use spring supplied jsp tag library tags they automatically take care of this process which is nothing but reading old values from command object and displaying in the form page.

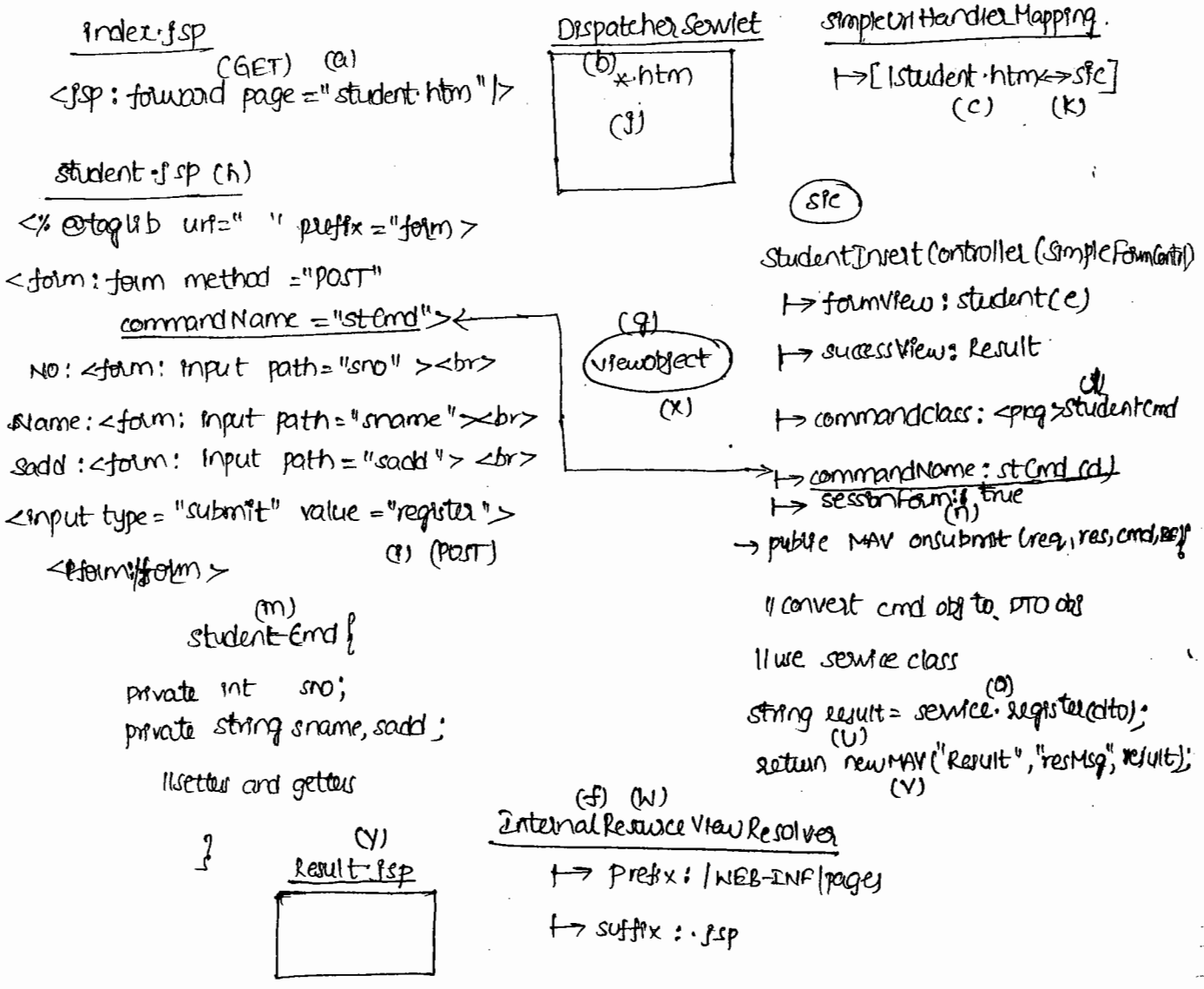
14/9/15

⇒ Spring gives two jsp tag libraries to develop that jsps of web application

a) Generic Jsp tag library
   - ↦ General tags
   - ↦ tld file: spring.tld
   - ↦ taglib uri: http://www.sf.org/tags

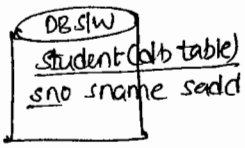
b) Form JSP tag library
   - ↦ Form tags
   - ↦ tld file: spring-form.tld
   - ↦ tag lib uri: http://www.sf.org/tagsform

Insert Usecase of Simple Form Controller



```
public class StudentInsertServiceImpl{
    public String register(dto){
    //convert DTO into BO
    ----
    int cnt = dao.insert(bo);
    if(cnt==0)
    return "Registration failed";
    else
    return "Registration succedded";
    }
}
```

```
public class StudentInsertDAOImpl{
    public int insert(BO){
    ---
    ---
    ---
    return o/1 (s);
    }
}
```



a→h : Initial Phase request

i→ : Post back request

MVCStudentApp(SimpleFormController)6

- ⊢→ JavaResources: src
  - ⊢→ com.nt.bo
    - ⊢→ StudentBO.java
  - ⊢→ com.nt.dto
    - ⊢→ StudentDTO.java
  - ⊢→ com.nt.command
    - ⊢→ StudentCmd.java
  - ⊢→ com.nt.controller
    - ⊢→ StudentInsertController.java
  - ⊢→ com.nt.service
    - ⊢→ StudentInsertService.java, StudentInsertServiceImpl.java
  - ⊢→ com.nt.dao
    - ⊢→ StudentInsertDAO.java, StudentInsertDAOImpl.java
- ⊢→ WebContent
  - ⊢→ index.jsp
  - ⊢→ WEB-INF
    - ⊢→ Pages
      - ⊢→ student.jsp
      - ⊢→ Result.jsp
    - ⊢→ dispatcher-servlet.xml
    - ⊢→ persistence-beans.xml
    - ⊢→ service-beans.xml
    - ⊢→ applicationContext.xml
    - ⊢→ web.xml

Jars

Spring 3.x

15/9/15

Form Validation in SimpleFormController

⇒ verifying the format and pattern of form data is called "Form Validation". SpringMVC allows to write form validation logic in seperate class so that it can be linked with multiple controller command classes having reusability.

Sample code

a) Develop validator class implementing Validator(I) and implement supports(-), validate(-,-) methods.

b) Develop Properties file having Validation error messages.

c) Configure the Properties file in dispatcher-servlet.xml

d) cfg validator class in dispatcher-servlet.xml and link that class with controller command class using the property "validator".

<property name = "validator" ref = "stValidator">

e) place <form: errors> path="*" > tag in student.jsp to display the form validation error messages. <form:errors path="*" />

⇒ The SimpleFormController can handle 3 types of validation

1) FormValidation(NO, Name cannot be blank)

2) TypeMismatch errors (storing string value to sno property)

3) Application logic errors (like not allowing delhi as address for student)

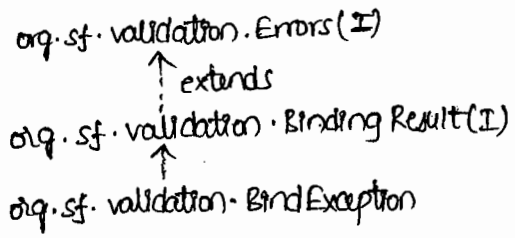
⇒ Instead of hard coding error message directly in validator class & controller class it is recommended to place in properties file having our choice keys and values. For TypeMismatch Errors the key should be "typeMismatch.<property name>."

16/9/15

Q What is the use of implementing supports(class clazz) method in validator class of SimpleFormController ?

A This supports(-) holds the cfg "commandclass" in clazz parameter and we can check whether correct command class configured or not by using clazz.IsAssignableFrom() method. If configure correctly then it returns true, so validate(-) method executes, if not configure correctly then it returns false. so validate(-,-) will not execute, due to this we can avoid mis configurations in spring bean configuration file.

Hierarchy of Errors Interface



Internal Flow of form Validations

⇒ Formpage submits the form (post back request) → DispatcherServlet traps the request and uses handler request to map the request to our simpleformcontroller and calls handleRequest(-,-). This method check whether request is "initialphase" or "postback req" since the request is post back request.

a) creates/locates command class object and writes form data to it. (requestwrapping)

b) creates an empty BindException

c) checks weather 'validator' is configured or not and notices if it is configured.

d) get validator object and calls supports(-) with class object that points command class if this method returns true then validate(-,-) will be called having errors object (BindException), and object otherwise validate(-,-) will not be called.

e) validate(-,-) performs form validations and stores validation Error in error object

f) handleRequest(-,-) checks the size of errors object to know whether validation errors are there or not there.

If Errors object size==0 (no form validation errors)

{

→ calls onSubmit(-,-,-) having req, res, cmd, BindException objects

→ onSubmit(-,-,-,-) will process the request

}

If Errors object size>0 (for form validation errors)

→ keeps errors object in request scope.

→ return MAV object having form view property value as logicalviewname, command class name, command class object as model attribute.

→ DispatcherServlet keeps the received command class object in request scope and uses view Resolver to get view object having form page physical view name.

→ Form page will be displayed in that <form:input> tags collect command object data to show old values in components and <form:error> uses 'errorNamespace object' data to show validations errors messages

}

Q What is double posting or duplicate form submission ? How can we solve the problem?

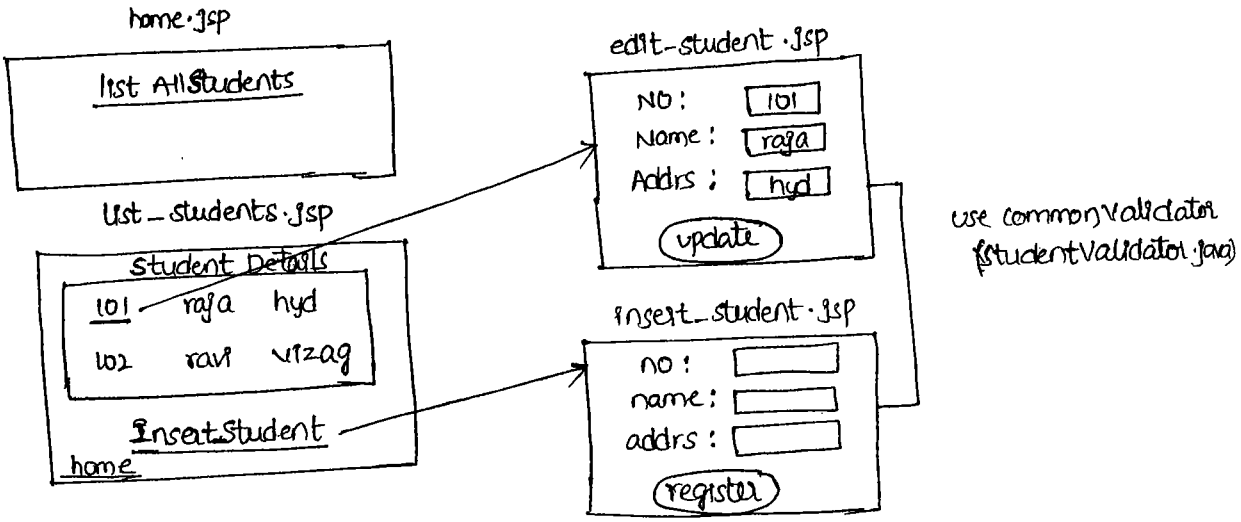
A Pressing refresh button on the result page of form submission submits the form one more time. clicking on submit button multiple times, continuously submits the form multiple times. This leads to duplicate form submission gives problems like duplicate registrations, deducting amount from dbt/ credit cards for multiple times and etc.. This is also called as "double posting problem."

⇒ We can overcome this problem with tokens support when form page is submitted one token will be created and will be maintain as session attribute once onSubmit(-,-) process the request it removes token from session object. If any request comes without token because of this double posting problem error will be raised.

⇒ In spring MVC, all these activities are handled internally if any request comes without token the handleInvalidSubmit(-,-) will be called so we can place logic to render error page in that method.

Refer page NO:21

17/9/15 Mini Project on Spring MVC



MVCMiniProjApp8

- javaResources
  - src
    - com.nt.bo
      - StudentBO.java
    - com.nt.dto
      - StudentDTO.java
    - com.nt.command
      - StudentCommand.java
    - com.nt.validator
      - StudentValidator.java
    - com.nt.dao
      - StudentDAO.java
    - com.nt.service
      - StudentService.java
    - com.nt.controller
      - ListStudentsController.java (AbstractController)
      - InsertStudentController.java (SimpleFormController)
      - UpdateStudentController.java (SimpleFormController)
- WebContent
  - index.jsp
  - WEB-INF
    - pages
      - home.jsp
      - edit_student.jsp
      - edit_result.jsp
      - dbinfo.jsp
      - list_students.jsp
      - insert_student.jsp
      - insert_result.jsp

⊢→ web.xml

⊢→ dispatcher-servlet.xml

⊢→ applicationContext.xml

⊢→ persistence-beans.xml

⊢→ service-beans.xml

⊢→ aop-beans.xml

18/9/15

Importance of formbackingObject method

⇒ Generally when handle request is processing. initial phase request. It internally formBackingObject method to create command class object. The predefined method always create empty cmd object. If we want to create cmd object with dynamic data according to application requirement then we need to override this method in our controller class that extends from simpleformcontroller.

Usecase: When we click on student id hyperlink from list of student details. If we want to display edit.jsp having that student details as initial values (refer MVCMiniprojApps appln).

20/9/15

referenceData (HttpServletRequest request)

⇒ While displaying form page as the response of initial phase request or as the response of post back request when form validation errors are there if we want to display additional data apart from the data that is there in the command object we can override referenceData(-) in our simpleFormController class to construct that data dynamically this Method return Map object having Data in the form of values with keys.

⇒ Dispatcher Servlet keeps this Map object in request scope. In order to display this Map obj data in certain component we need to specify that Map object in the form of "items" attribute of <form:options>, <form:checkboxes> and etc.. tags.

Ex

```
public class MyFormController extends SimpleFormController {
    Public Map referenceData(HttpServletRequest req){
        Map<String, List> map = new HashMap();
```

```
        List list = new ArrayList();
            list.add("T.L");
            list.add("P.L");
            list.add("G.L");
            ----
        map.put("rolesList", list);
        return map;
    }
}
```

In Form Page.

```
<form:select path="role">
    <form:options items="${rolesList}"/>
</form:select>
```

Usecases: Displaying roles in form page (select box)

⇒ for initial phase request and in post back request when it generates form validation errors.

For reference: MVC Ref DataApp9

```
├→ src
│   ├→ com.nt.command
│   │   ├→ RegistrationCommand.java
│   ├→ com.nt.controller
│   │   ├→ StudentRegistrationController.java
│   ├→ com.nt.validator
│   │   ├→ StudentInsertValidator.java
│   │   ├→ stCmdValidationErrors.properties
├→ webcontent
│   ├→ index.jsp
│   ├→ WEB-INF
│   │   ├→ pages
│   │   │   ├→ register.jsp
│   │   │   ├→ register_result.jsp
│   │   ├→ web.xml
│   │   ├→ dispatcher-servlet.xml
│   │   ├→ applicationContext.xml
```

InitBinder() Method

⇒ Property Editors of spring are very useful to convert given string values to required type properties of bean class (or) command class.

⇒ Spring gives built-in property editors and allows us to develop custom property editors. CustomDateEditor is pre-defined property editor class that can be used to convert given string date values into java.util.Date class object by specifying date format.

⇒ While working with SimpleFormController we can override initBinder() method to register propertyEditors that are required to use while binding given form data (request parameter values to command class object properties). This method gives request, ServletRequestDataBinder object. that can be used we can register property editors with ServletRequestDataBinder object because it is actually responsible to bind form submitted request parameter values to command object properties.

```
@Override
protected void initBinder(HttpServletRequest request, ServletRequestDataBinder binder)
                                                    throws Exception{

    SimpleDateFormat sdf = new SimpleDateFormat("dd-MM-yyyy");

    binder.registerCustomEditor(Date.class, new CustomDateEditor(sdf,true));

}
```

usecase: storing string values of form components to command class properties as java.util.Date class object.

Internal flow of execution in the handleRequest method of SimpleFormController

1. Initial Request

Form Display

GET Request

Call Sequence

1. formBackingObject(-) (cmd object creation)
2. initBinder(-,-) (property editor registration)
3. showForm(-) (to show formpage based on formView)
4. referenceData(-) (to carry additional data required for form page)

Request 1

(Form GET)

2. Invalid Form Submission

POST Request

Call Sequence

1. initBinder(-,-)
2. onBind(-) (to bind form data to cmd object)
3. onBindAndValidate(-)
4. processFormSubmission(-)
5. showForm(-)
6. referenceData(-)

Request 2

(Failed Submission)

3. Valid Form Submission

POST Request

Call Sequence

1. initBinder(-,-)
2. onBind(-)
3. onBindAndValidate(-)
4. processFormSubmission(-)
5. onSubmit(-)

Request 3

(Successful Submission)

Refer : ~~MIXE~~ ~~APP10~~ MVC DataBinderSPC App10.

21/9/15

MultiActionController

When form page contains multiple submit buttons then instead of taking multiple abstractCommandController|SimpleFormController classes we can take single MultiActionController class having multiple user-defined methods to process requests of multiple submit buttons on 1 per method basis.

⇒ The user-defined methods of this class can have following standard signature

public (ModelandView|Map|String|void) <methodName> (HttpServletRequest request, HttpServletResponse response, [HttpSession], [AnyObject]);

⇒ While configuring our MultiActionController class we must specify MethodNameResolver object as dependent object that contains additional parameter name based on which the method that has to be executed in MultiActionController class will be decided.

```
<bean id="mnr" class="org.sf.web.servlet.mvc.multiaction.ParameterMethodNameResolver">
   <property name="paramName" value="opt" />
   <property name="defaultMethodName" value="invalid" />
</bean>

<bean id="poc" class="com.nt.controller.PerformOperationsController" >
                                  ↳ MultiActionController
   <property name="methodNameResolver" ref="mnr" />
</bean>
```

User.jsp

```
<form action="perform.htm" method="post">
  ------
  ------
  ------
<input type="submit" value="insert" name="opt" >
<input type="submit" value="update" name="opt" >
<input type="submit" value="delete" name="opt">
<input type="submit" value="view" name="opt1">
</form>
```

PerformOperationController.java

```
public class PerformOperationController extends MultiActionController {
    public ModelAndView insert(req, res, cmd){
        ---
    }
    public ModelAndView update(req, res, cmd){
        ---
    }
    public ModelAndView delete(req, res, cmd){
        ---
    }
    public ModelAndView invalid(req, res, cmd){
        ---
    }
}
```

Conclusion

⇒ If source page is form submission having multiple submit buttons with optional form validations and the required destination page is static/dynamic webpage that should come after request processing then we should go for MultiActionController.

Flow for insert submit button

insert submit button → DispatcherServlet → HandlerMapping → Controller (MA Controller) → handleRequest(-,-) →

a) creates Command object and perform request wrapping

b) reads paramName (opt), defaultMethodName (invalid) from the injected MethodNameResolver Obj

c) reads the value of "opt" reqest param (insert)

d) calls insert (req, res, cmd/obj).

Example

MVCMultiActionControllerApp11

- Java resources
  - src
    - com.nt.controller
      - PerformOperationsController.java
    - com.nt.command
      - UserCommand.java
  - WebContent
    - index.jsp
    - WEB-INF
      - dispatcher-sewlet.xml
      - application-context.xml
      - web.xml
      - pages
        - user.jsp

Jars: Spring 4.x jars

Note only SimpleFormController is removed from spring 4.x, Remaining all controller are there

Refer: Mvc App on MultiActionController

Note While working with MultiActionController there is no need of configuring command class anywhere because the super class handleRequest() internally uses reflection api to take specified third parameter type or fourth parameter type (when third parameter is HttpSession or fourth parameter type as command class name).

22/9/15 **AbstractWizardFormController**

⇒ So far we working with single form page based controller classes like SimpleFormController, MultiActionController etc.. In order to work with multiple form page that comes in a sequence having wizard style we need to use AbstractWizardFormController by specifying list of form pages to display in sequence. In this controller the data of all the form pages will come and store in a single command class object. Based on name of additional request parameter that comes along with the request this will display form page (or) process the request.

- _cancel : leaves the wizard with out performing final action (cancel button)
- _finish : leaves the wizard by performing final action (finish button)
- _target<x> : Displays the nth page from the list of form pages that are configured. (for previous, next button).

Welcome page → Form page1 → Form page2 → form page3 ... → result page

⇒ We can override the following methods in our AbstractWizardFormController

a) formBackingObject(-) : creates and returns command object

b) processFinish(--) : Takes final action of request processing for "finish" button.

c) processCancel(--) : executes the logic related to "cancel" button.

Application Flow

index.jsp (a)

```
<jsp:forward page="home.htm">
```

DispatcherServlet

*.htm (b) (1) (7) (2) (a8)

SimpleUrlHandlerMapping

(c) → home.htm → PVC1
→ register.htm → rc (3) (5) (a1) (a9)

(PVC1) **ParameterizableViewController**

→ viewName : Welcome (d)

(rc) (4) (6) (a2) (a10)

RegisterController (AWFController)

→ pages (m)
→ page1Form (o)
→ page2Form (u)
→ page3Form (a2)
→ commandName : regCmd
(l) formBackingObject(-){
return new RegisterCommand;
} (a11)
processFinish(req, res, cmd, BE){
--return new MAV(-,-,-); (a12)
}
processCancel(req, res, cmd, BE){
--return new MAV(-,-,-);
}

Welcome.jsp (g)

```
<a href="register.htm">
    register </a> (h)
```

page1Form.jsp (p)

```
<form:form method="POST">
---
<input type="submit" name="_target1"
        value="next"> (q)
</form:form>
```

(f) (o) (n) (a5) (a14) (ViewObject)

page2Form.jsp (x)

```
<form:form method="POST">
---
<input type="submit" name="_target2" (y)
        value="next">
<input type="submit" name="_target0"
        value="previous"><input type="submit" name="_cancel"
```

(e) (n) (v) (a4) (a13)

InternalResourceViewResolver

→ prefix : /WEB-INF/pages
→ suffix : .jsp

page3form.jsp (a6)

```
<form:form method="POST">(a7)
<input type="submit" name="_finish"
                     value="finish">
<input type="submit" name="_target1" value="previous">
<input type="submit" name="_cancel" value="cancel">
</form:form>
```

result.jsp (a15)

= = = =

MvcWizardFormApp12

- javaresource
  - src
    - com.nt.controller.
      - RegisterController.java
    - com.nt.command
      - RegisterCommand.java
- webcontent
  - index.jsp
  - WEB-INF
    - pages
      - welcome.jsp
      - page1form.jsp
      - page2form.jsp
      - page3form.jsp
      - result.jsp
    - web.xml
    - dispatcher-servlet.xml
    - application-context.xml

Refer: App(10) of page32

Note

⇒ When we give initial phase (GET) request to AbstractWizardFormController it initially called formBackingObject() to create command object and to keep that object in session scope.

⇒ When post back request is given having _target(x) request parameter name it displays next or previous form in the list of pages that are configured.

⇒ For post back request having request parameter name having "_cancel" calls process cancel() through handle request similarly calls processFinish() for post back request that contains "_finish" request param.

23/9/15

Handler Mapping.

⇒It is component that maps incoming request to controller/ handler class. All Handler mapping classes are the implementation classes of 'HandlerMapping' interface.

⇒The important handler mapping classes are

a) BeanNameUrlHandlerMapping (Default for xml configurations)

b) ControllerClassNameHandlerMapping

c) DefaultAnnotationHandlerMapping

d) SimpleUrlHandlerMapping

a) BeanNameUrlHandlerMapping

⇒ This maps incoming request to controller class based on the servlet path/ virtual path of incoming request urls, the virtual path of incoming request url must be taken as bean name of controller class.

Ex

```
<bean class="org.sf.web.servlet.handler.BeanNameUrlHandlerMapping">
<bean name="/home.htm" class="com.nt.controller.WelcomeController"/>
```

If the incoming request url contains "/home.htm" then it maps the request to WelcomeController class.

b) SimpleUrlHandlerMapping

⇒This is useful to map incoming requests to with controller classes based on virtual paths of request url's and bean id's/*names of controller classes. We need to supply these details as the values of "mappings" property (of type java.util.Properties).

Ex:

```
<bean class = "org.sf.web.servlet.handler.BeanSimpleUrlHandlerMapping">
  <property name = "mappings">
     <props>
       <prop key = "/home.htm">wc</prop>
     </props>
  </property>
</bean>
<bean name = "wc" class = "com.nt.controller.WelcomeController"/>
```

⇒ If request url contains "/home.htm" then it passes the request to welcomecontroller class whose beanname "wc".

c) ControllerClassNameHandlerMapping

⇒ Maps incoming requests with controller classes based on virtual paths and controller class names. It takes virtual path & removes extension, makes first letter as upper case letter and appends controller word to it meanwhile looks for controller class whose name is that name.

```
<bean class="com.nt.controller.WelcomeController"/>
<bean class="org.sf.web.sewlet-mvc.suppot.ControllerClassNameHandler"/>
```

⇒ If request url contains "welcome.htm" then it maps the request to a controller class whose name is "WelcomeController".

d) DefaultAnnotationHandlerMapping

⇒ Maps the incoming requests url to controller classes methods based on @RequestMapping annotation information.

Ex

```
@Controller
public class MyController {

   @RequestMapping {"/home.htm"},
  public MAV m1(){
        ---
        ---
      }
  @RequestMapping {"/welcome.htm"}
   public MAV m2(){
        ---
        ---
      }
```

⇒ If request url contains "/home.htm" then request goes to m1() method, if request url contains "/welcome.htm" then request goes to m2() method.

Note When we configure multiple handler mappings in which order then should be executed will be decided based on order priority value. High value indicates low priority and low value indicates high priority.

```
<bean class="org.sf.web.sewlet.handler.SimpleUrlHandlerMapping">
  <property name="mappings"> <props> <prop key="welcome.htm">WC</prop></props></pro
  <property name="order" value="1"/>
</bean>
<bean class="org.sf.web.sewlet.mvc.support.ControllerClassNameHandlerMapping">
  <property name="order" value="2" />
</bean>
```

⇒ While configuring HandlerMappings we can also specify "order" and "handlerInterceptor".

HandlerInterceptor

⇒ It acts as hook for controller/handler class to place have pre and post processing logics. It is like ServletFilter of JEE web application development but it is different filter in 1 way. Filter can have only pre-request processing and post-response processing logics but HandlerInterceptor can handle pre-request processing, before rendering view and after rendering view logics with respect to controller classes.

⇒ To develop interceptor we need to take a class that implements "HandlerInterceptor(I)" but there is adaptor class for this interface whose name is "HandlerInterceptorAdapter", so we can extend from this class to override our choice methods.

preHandle(--) → executes before controller goes to the controller class

postHandle(--) → executes once controller class finishes the execution but before rendering the view.

afterCompletion(--) → executes after rendering the view and etc..

25/9/15

HandlerInterceptor usecases

⇒ We can develop handle interceptor class for controller class only to take the requests between 9am to 5pm

⇒ we can develop handler interceptor class for controller only to allow request that are coming from certain browser and etc..

For example

⇒ keep any spring mvc appln ready as seperate App (MVC second web).

⇒ Develop HandlerInterceptor class in com.nt.interceptor package as shown below,

Appln 9 of MVC page NO: 30.

ViewResolvers

⇒This component takes logical viewnames given by controller through DispatcherServlet and returns "viewobject" (The implementation class object of view(I) pointing to physical view name. Dispatcher servlet class render(-,-) method on this view object to render the control to view resource like html file, jsp file, bean class that gives PDF, Excel and etc..docs.

⇒ Every ViewResolver is a java class that implements "viewResolver"(Interface) spring hay supplied few pre-defined View Resolver classes. They are

a) InternalResourceViewResolver : Useful to resolve jsp, servlet programs of WEB-INF folder as views. By default takes "JstlView" as the view class.

b) UrlBasedViewResolver : super class for (a) and we must specify view class explicitly

c) ResourceBundleViewResolver : Allows to cfg views in properties file.

d) XmlViewResolver : Allows to configure views in xml file.

e) BeanNameViewResolver : useful to resolve spring Beans as views and etc...

a) UrlBasedViewResolver

⇒Takes logical view name from controller class and searches for the view resource whose name is same as logical view name. We must configure "view class" explicitly while configuring this ViewResolver class.

⇒ This ViewResolver can be used to render any view like html, jsp, bean class

⇒ We configure JstlView as view class to make view Resolver to resolver html|jsp|servlet programs as views that means viewResolver gives JstlView class object and render method will be called on that object to render the view to that object.

```
<bean class="org.sf.web.servlet.view.UrlBasedViewResolver">
 <property name="viewClass" value="org.sf.web.servlet.view.JstlView"/>
 <property name="prefix" value="/WEB-INF/pages"/>
 <property name="suffix" value=".jsp"/>
</bean>
```

b) InternalResourceViewResolver

⇒It is sub class of UrlBasedViewResolver that takes JstlView as the default view class having capability to ~~render~~ resolve the sewlet, jsp programs of webapplication that are these under WEB-INF folder. It is most regularly used viewResolver.

```
<bean class="org.sf.web.servlet.view.InternalResolverViewResolver" >
    <property name="prefix" value="/WEB-INF/pages" />
    <property name="suffix" value=".jsp" />
</bean>
```

c) ResourceBundleViewResolver

⇒The problem with above two resolvers is logical view name and jsp file name must match. If the jsp file name is changed later, then modifying logical view name by opening the source code of controller class is very difficult.

⇒In order to overcome this problem we can use ResolverBundleViewResolver where we can configure views in the properties file.

views.properties (src) (default name is also views)

welcome.(class) = org.sf.web.sewlet.view.JstlView → If views are jsp's we write this class.

welcome.url = /WEB-INF/pages/welcome.jsp
↳ logical view name

Note: We have change class names based on our requirement.

Indispatcher-sewlet.xml

```
<bean class="org.sf.web.sewlet.view.ResourceBundleViewResolver" >
    <property name="basename" value="views" />
</bean>
```

26/9/15
Note We can use ResourceBundleViewResolver to enable Internationalization in spring mvc appln. we can take multiple jsp's for multiple locales we can configure them in multiple locales configuration files.

d) XmlViewResolver

⇒Similar to ResourceBundleViewResolver but allows to take xml file to specify views configuration. Takes views.xml as the default name. we need to specify this file name as the value of "location" property while configuring "XmlViewResolver".

views.xml (WEB-INF folder)

```
<bean id="welcome" class="org.sf.web.sewlet.view.JstlView">
<property name="url" value="/WEB-INF/pages/welcomehome.jsp"/>
</bean>
```

dispatcher-servlet.xml

```
<bean class="org.springframework.web.servlet.view.XmlViewResolver">
<property name="location" value="/WEB-INF/views.xml"/>
</bean>
```

e) BeanNameViewResolver

⇒ Resolves java class/ spring bean as the view resource, for this the logical view name gathered from the controller class must be taken as bean id (or) name for the class/bean that is acting as view.

For example appln on various ViewResolvers refer: 12 page no: 37

dispatcher-servlet.xml

```
<!--Java class/ Spring bean acting as view-->
<bean name="welcome" class="com.nt.view.MyReportExcel"/>
                    ↳ logical view name for java class.
<bean id="vr" class="org.sf.web.servlet.view.BeanNameViewResolver"/>
```

Note when we configure multiple ViewResolver they execute in a chain. InternalResourceViewResolver always gets high priority when compared to other ViewResolver in chain. so it is always recommanded to configure InternalResourceViewResolver as the last resolver in the chain.

Working with Views

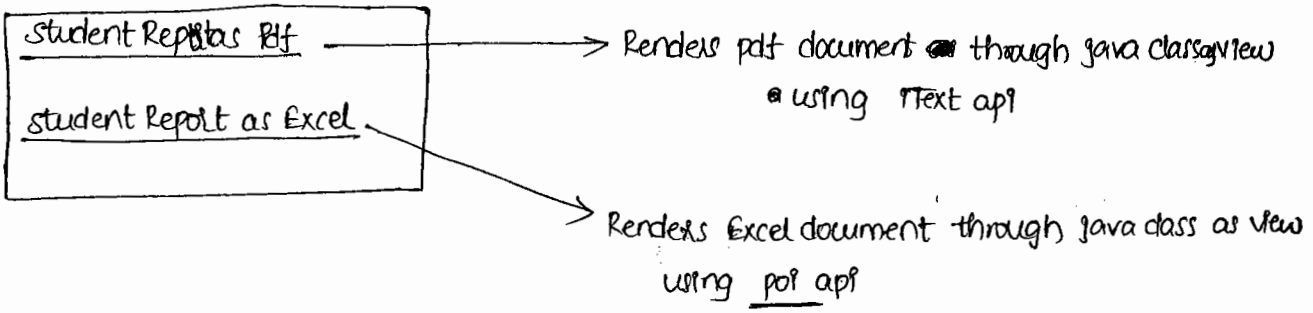
⇒ Spring allows to take different technologies in the view layer like jsp, html, servlet, velocity, free marker etc.. In spring we can take java classes/ spring beans as views by extending them from AbstractXxxView classes like (AbstractPdfView, AbstractExcelView,..) to render response as pdf, excel and etc. documents.

⇒ In spring, view is an abstract entity so we can take any thing as view.

⇒ ViewResolver gives "view object" (View(I) implementation class object) and DispatcherServlet calls render(-) on the object. This render(-) method passes control to physical view file/class (like jsp on the above discussed classes).

Example Appln on BeanNameViewResolver and Java classes as views

home.jsp



MVCBeanNameViewResolverApp16

- JavaResources
  - src
    - com.nt.controller
      - ListStudentController.java
    - com.nt.view
      - StudentExcelView.java
      - StudentPdfView.java
- WebContent
  - index.jsp
  - WEB-INF
    - pages
      - home.jsp
    - web.xml
    - dispatcher-servlet.xml
    - application-context.xml

springmvc + standard.jar + jstl.jar + IText-2.0.7.jar (for pdf) + poi-3.0.1.jar (for Excel)

↳ from spring 2.5 version.

2/4/15

I18N (Internationalization)

⇒ Making our application working for different locales by rendering labels in different languages is called enabling I18N on the application. This improves customer/visitors to webapplication. For this we need to make multiple properties files for multiple locale on 1 per locale basis like welcome.properties, welcome_fr_FR.properties, welcome_zh_CN.properties (base file) and etc..

⇒ To enable I18N on spring mvc appln we need to configure LocaleResolver, LocaleChangeInterceptor classes as beans and we need to pass interceptor reference to HandlerMapping class configuration to like interceptor with controller class.

⇒ SessionLocaleResolver is implementation class of LocaleResolver interface that allows cfg locale explicitly otherwise fallback to default locale cfg and accept-language header value.

```
<bean id="localeResolver" class="org.sf.web.sewlet.i18n.SessionLocaleResolver">
<property name="defaultLocale" value="en"/>
</bean>
```

⇒ LocaleChangeInterceptor allows to change locale for every request with respect to controller class based on the additional request parameter that is configured.

```
<bean id="lci" class="org.sf.web.sewlet.i18n.LocaleChangeInterceptor">
    <property name="paramName" value="language"/>
</bean>
```

⇒ Based on the value of language request parameter the Locale will be changed for every request by overriding default locale. Based on this locale the property file will be picked up to render the labels.

⇒ While preparing multiple properties files in the appln of web I18N all properties files must have same keys and different labels specific to each locale.

Example Appln page no: 42

MVCI18NAppl7.

- Java Resources
  - src
    - com.nt.controller
      - WelcomeController.java
    - welcome.properties
    - welcome_zh_CN.properties
- WebContent
  - redirect.jsp
  - WEB-INF
    - pages
      - Register.jsp
    - web.xml, dispatcher-servlet-application-context.xml

Annotation-driven Spring MVC

⇒In xml cfg based programmatic approach of spring mvc we develop controller classes by implementing spring specific interfaces and by extending from spring specific classes. This kills

It also make the programmer to implement or override with specific/fixed methods signatures where programmer will be forced to place unnecessary parameters like request, response eventhough he is not using them. Same thing can be said about return types of the methods.

⇒To develop controller classes as POJO classes having flexibility of placing user-defined methods with user-defined signatures we need to go for Annotations driven spring MVC. These annotations are introduced from spring 2.5.

⇒All new projects and in the migration projects annotations driven spring mvc programming is required. Two implementations are

@Controller → cfg java class as spring MVC controller class and spring bean

@RequestMapping → To map incoming request urls with the methods of controller class based on the virtual path of request url.

28/9/15

⇒While working with annotation based spring MVC controller we can place flexible methods having flexible parameter types and flexible return types

```
@Controller
public class MyController {
 @RequestMapping ("home.htm")
public <--> methodname (--) {
  ---
 }
```

Important possible parameter types

HttpServletRequest, HttpServletResponse, HttpSession, Command class

@ModelAttribute

BindResults/Errors

Map

ModelMap

Model

and etc- (lots of other params supporting webservices)

Important possible return types

- ⇒ void
- ⇒ ModelAndView
- ⇒ String
- ⇒ Model
- ⇒ Map
- ⇒ View

and etc.. (lots of other return types supporting webservices)

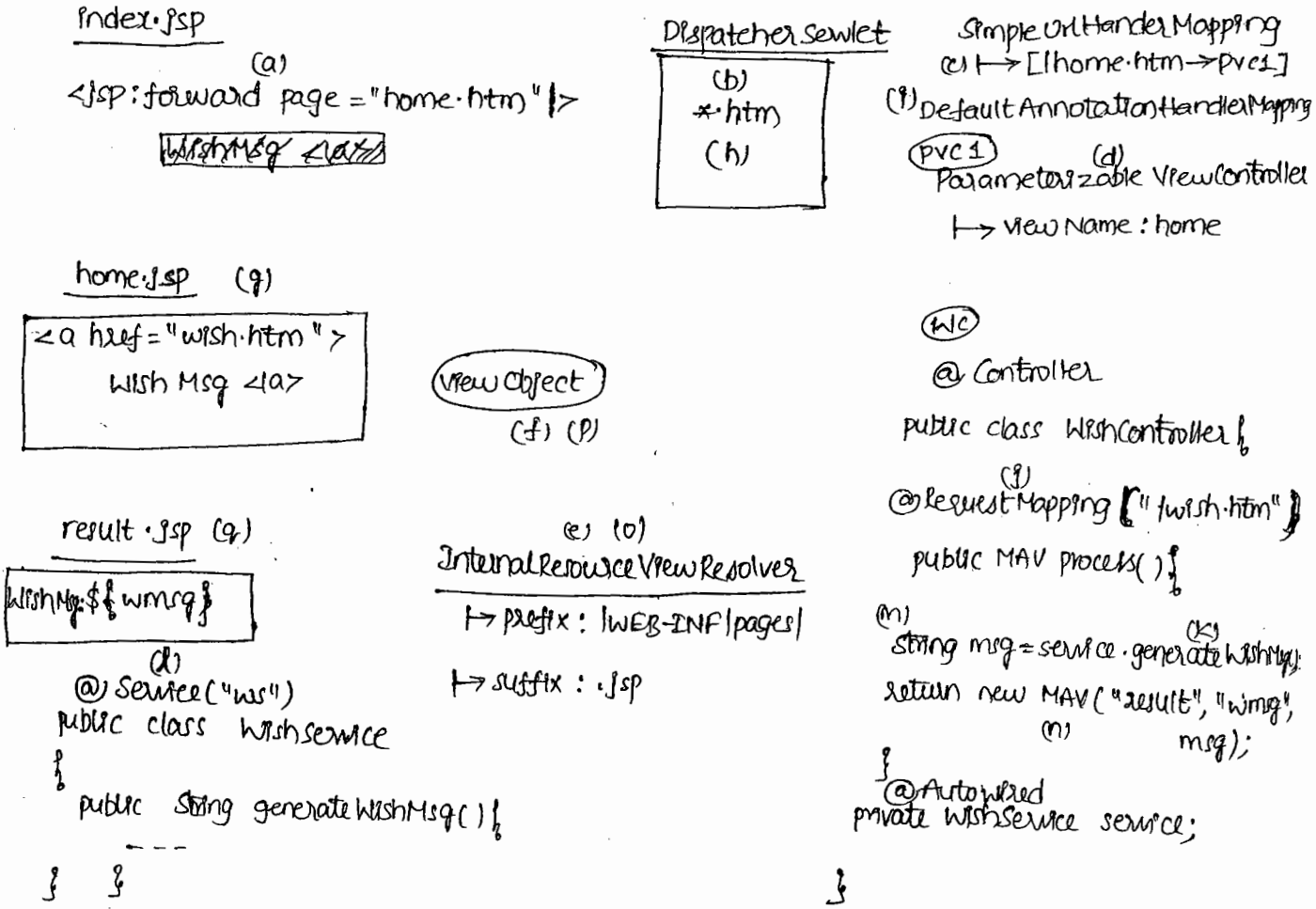
Note In spring MVC only controller classes can be configured with annotations and the remaining components like handler mappings, view resolvers and etc.. must be configured through xml files.

Abstract Controller (using annotations)

⇒ Source page : NO form submission

⇒ Destination page : static/dynamic form page with request processing.

Note We still use built-in forwarding control like ParameterizableViewController using xml configuations.



MVC-Annt WishApp 18 (Abstract Controller)

```
|→ javasources
     |→ src
          |→ com.nt.controller
               |→ WishController.java
          |→ com.nt.service
               |→ WishService.java
|→ webcontent
     |→ index.jsp
     |→ WEB-INF
          |→ pages |→ home.jsp, result.jsp.
          |→ web.xml
          |→ dispatcher-servlet.xml, application-context.xml
```

Jars : same as spring MVC + AOP Lib jars

refer page NO:41 & app no :15

29/9/15

RequestToViewNameTranslator

⇒If controller class handler method (annotated with @RequestMapping) return type is void (ie, not returning logical view name) then DispatcherServlet takes request url (virtual path) of current request and translates into view name based on "org.sf.web.servlet.view.RequestToViewNameTranslator".

In Controller class

```
@RequestMapping("/wish.htm")
public void process(Model model){
---
model.addAttribute("wmsg",msg);
}
```

If the request url is→"http://localhost:3030/WishAppl/wish.htm" then "wish" will be taken as logical view name and wish.jsp will be taken as view resource if the resolver in InternalResourceViewResolver. For this DispatcherServlet internally uses "DefaultRequestToViewNameTranslator". This translator removes prefix leading and trailing slashes and also removes extension from request url (makes "/home.htm" as "home") to use the result as logical view name.

Ex: refer page NO:40 of 13)MVC App on RequestToViewNameResolver.

Annotation based AbstractCommandController

Source Page

→ Form page with form submission (command class is required)

→ No server side form validation

→ No form handling

Destination Page

→ static / dynamic page through request processing

⇒ Use @ModelAttribute based param in handler method of controller class to specify the command class & its object to perform request wrapping [writing form data into command class object]

⇒ @ModelAttribute use as parameter in handler method.

Web application Context

index.jsp (a)

```
<jsp:forward page="performsearch.htm">
```

DispatcherServlet

```
*.htm
(b)
(i)
```

↦ SimpleUrlHandlerMapping

↦ [performsearch.htm] ⇔ pvc1 (c)

(j)

↦ DefaultAnnotationHandlerMapping

pvc1 (d)

ParameterizableViewController

↦ viewName : search

search.jsp (g)

```
<form action="performsearch.htm">
No: <input type="text" name="no"/>
---
---
<input type="submit" value="search" />
(h)
</form>
```

ViewObject (p) (f)

@Controller

```
public class EmpSearchController {
    (k)
    @RequestMapping("/performsearch.htm")
    public String search(ModelMap map,
        (m) @ModelAttribute
            SearchCmd cmd)
    {
    ---
    ---
    return "result";
    (n)
    }
```

↳ Recognizes SearchCmd as the command class.

(e) (o)

InternalResourceViewResolver

↦ prefix : /WEB-INF/pages/

↦ suffix : .jsp

SearchCmd.java

```
public class SearchCmd {
    private int no, salary;
    private String name, desg;
    // setters & getters
    (l) → request wrapping
}
```

result.jsp

```
Display results
```

MVC-AnnoEmpsearchApp-19(AbstractCommandController)

- Javaresources
  - src
    - com.nt.controller
      - EmployeeSearchController.java
    - com.nt.command
      - SearchCmd.java
    - com.nt.dto
      - EmployeeDTO.java
- webcontent
  - index.jsp
  - WEB-INF
    - pages
      - search.jsp
      - emplist.jsp
    - web.xml
    - application-context.xml
    - dispatcher-servlet.xml

Jar files : same

Refer page NO: 46 of App16.

Skills

Annotations based SimpleFormController

Source page

Form page → form submission
Form validations (server side)
Form handling

Destination page

static/Dynamic page that comes after request process

Initial phase request (req method == GET)

→ creates command object
→ InitBinder
→ adds command obj to model attribute
→ reference data
and etc...

Postback request (req method==POST)

- → creates / locates command object
- → init binder
- → BindException object creation
- → validator, supports (-), validate (-,-)
- → reference Data
- → onSubmit (-,-)

and etc....

Web application context container

index.jsp (a)

```
<jsp:forward page="login.htm"/>
```

DispatcherServlet

```
*.htm
(b) (j)
```

login.jsp (h) (v)

```
<form:form method="POST"
---
<input type="submit" value="login"> (i)
</form>
```

View Object (g) (u)

(k)

Command class

```
public class User{
String user, pwd;
---
}
```

(f) (t)

InternalResourceViewResolver

- ↦ prefix : /WEB-INF/pages/
- ↦ suffix : .jsp

(c)

DefaultAnnotationHandlerMapping

```
@Controller
public class LoginController
{
        Initial phase Request
@RequestMapping (value="/login.htm",
        method = RequestMethod.GET)
(d)
public String showForm(Model model){
    User user = new User();
    model.addAttribute("userCmd", user);
    return "login";(e)
}
        postback request
@RequestMapping (value="/login.htm",
        method=RequestMethod.POST)
(l)
public String processRequest (Model model,
    @MethodAttribute ("userCmd") User user,
    BindingResults errors) {
---
                (m)
String msg = service.authenticate(userDTO);
return "login";
```

Application Context container

```
@Service ("ls")
public class LoginService {
    public String authenticate(userDTO){   (n)
    ---
(r) int cnt = dao.validate (userBO);   (o)
        if (cnt==0)
          return "Invalid credentials";
        else
          return "valid credentials";
    }
}
```

```
@Repository (s)
public class LoginDAO
{
  private JdbcTemplate jt;
                 (p)
  public int validate (UserBO){
  ---
  ---
  return cnt;
        (q)
}
```

MVC - AnnoLoginApp80 (SimpleFormController)

```
|-> JavaResources
    |-> src
        |-> com.nt.controller
            |-> LoginController.java
        |-> com.nt.service
            |-> LoginService.java
        |-> com.nt.dao
            |-> LoginDAO.java
        |-> com.nt.dto
            |-> UserDTO.java
        |-> com.nt.command
            |-> User.java
        |-> com.nt.bo
            |-> UserBO.java
        |-> com.nt.validator
            |-> LoginValidator.java
        |-> com.nt.commons
            |-> errors.properties
|-> WebContent
    |-> index.jsp
    |-> WEB-INF
        |-> pages
            |-> login.jsp
        |-> web.xml, applicationContext.xml, persistence-beans.xml,
        |-> dispatcher-Servlet.xml, service-beans.xml
```

Jars : MVC + JDBC + jstl + javax.inject.jar

Aim:

⇒ To perform form validations while working with Annotation based SimpleFormController

(i) prepare properties file having form validation error messages

(ii) Configure properties file

(iii) Develop validator class implementing Validator interface and overriding supports(), validate() method.

(iv) Inject validator object to controller class and call supports, validate method in the handler method that can process post back request.

(v) place <form:errors> tag in form page to display form validation errors.

⇒ While working with annotations based SimpleFormController we can use @ModelAttribute annotated method as alternate to reference data method of xml based SimpleFormController this holds data that has to be displayed in the form page when form validation errors are generated.

In controller class

```
@ModelAttribute("domains")
public List<String> populateDomains(){
    List<String> domainsList = new ArrayList<String>();
      domainsList.add("gmail.com");
     domainsList.add("yahoo.com");
     domainsList.add("rediff.com");
    return domainsList;
}
```

In form page

```
Domains : <form:checkboxes items="${domains}" path="dmn" />
```

(${domains} → ModelAttribute name; path="dmn" → command property.)

⇒ In order to register propertyEditor that are required to convert form supplied string values to various types of command class property we override initBinder() method in our controller class while working with xml cfg based SimpleFormController.

⇒ We can bring the same effect by using @InitBinder while using annotations driven SimpleFormController.

```
@InitBinder
public void myDateBinder(WebDataBinder binder){
    SimpleDateFormat sdf = new SimpleDateFormat("dd-MM-yyyy");
        binder.registerCustomEditor(Date.class, new CustomDateEditor(sdf,true));
}
```

File Uploading & file downloading

⇒ The process of selecting file from client machine filesystem and sending that file to server machine file system is called File Uploading and reverse is called File Downloading.

⇒ while developing matrimonium, job portal, video sharing the file uploading & downloading quite required.

⇒ Spring MVC gives built-in support for file uploading.

⇒ We need to configure "CommonsMultipartResolver" in dispatcher-servlet.xml to make controller class ready to receive Multi MIME type parts based form data (form data having uploaded files). Using this class we can also configure properties like "maxUploadSize", "MaxInMemory Size", "default Encoding" properties.

Perform the following operations for file uploading

⇒ Take file uploading component in form page also specify "enctype" and request method as "POST".

```
<form:form method="post" enctype="multipart/form-data">
    select file <input type="file" name="file" /> <br>
    ---
</form>
```

⇒ Take command class having property of type "MultipartFile" to receive & hold the uploaded file.

⇒ Configure "CommonsMultipartResolver" class as bean in dispatcher-servlet.xml file.

⇒ write streams based logic in controller class to save the received file in server machine file system.

Refer appln NO: (18) page NO: 54.

Notes

⇒ The process of getting the file belonging to server machine filesystem to client machine file system is called "Filedownloading".

⇒ By using content: Disposition Header we can instruction to browser to display the generated output as downloadable file.

⇒ We can use @RequestParam annotation to store the received request parameter into handler Method parameter.

⇒ Example appln NO: (19) page NO: 57 to 59.

3/10/15

Spring MVC with tiles

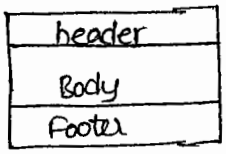
⇒ A tile is a logical portion (or) partial partition in a webpage like header tile, footer tile, body tile and etc..

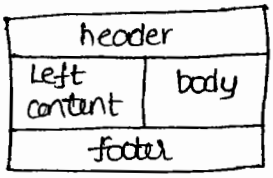
⇒ Apache tiles framework (plugin) is given to construct web pages based on layout page having layout control. This is like working with Composite View Design pattern.

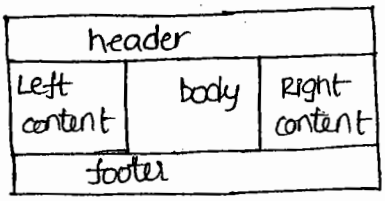
⇒ Layout control means all the web pages will be constructed based on single layout page for uniform look and any modification in layout page will reflect to all the webpages.

⇒ popular layouts are

a) classic layout b) Two-column layout c) 3 column layout d) circular layouts







⇒ Spring gives built-in support to work with tiles, for this we need to configure "TilesConfigurer" specifying tile definition file name (xml). The default file name is "WEB-INF/tiles.xml".

⇒ Every tile definition specifies the tile values in each webpage based on layout page. we create tile definition on 1 per webpage basic having logical view name. We can use "TilesViewResolver" to display webpage based on the each file definition.

Procedure to develop Spring MVC with the tiles application

a) Design layout page for templating webpages and to maintain webpages having layout control

b) Develop tiles.xml in WEB-INF folder having tile definitions based on no.of webpages in our website on 1 per tile definition basis. We can use tiles definition inheritance concept here.

c) Configure "TilesConfigurer" class in dispatcher-servlet.xml file as spring bean specifying "tile definition cfg file" file.

d) Configure "TilesViewResolver" as view resolver

e) Make controller class handler methods to return "tile definition name as logical view name"

⇒ For the above steps based complete example appln refer page NO : 59 (Spring MVC with tiles 2)

MVC - AnnoTiles App With Multi Action Controller - 23.

- Java resources
  - src
    - com.nt.controller
      - HomeController.java
- webcontent
  - redirect.jsp
  - WEB-INF
    - pages
      - *.jsp
    - dispatcher-servlet.xml
    - web.xml
    - tiles.xml
    - applicationContext.xml

Jar files: MVC jars + AOP lib

+

Tiles jar: tiles-template.<ver>
tiles-servlet.<ver>
tiles-jsp.<ver>
tiles-core.<ver>
tiles-api.<ver>
slf4j-1.2.<ver>
slf4j.<ver>
antlr.<ver>
commons-collections, common-beanutils, commons-digester,
commons-io, commons-fileupload.

Spring ORM

⇒ Jdbc technology allows the programmers to develop DB slw dependent persistence logic by using sql Queries due to this jdbc code is not portable across the multiple DB slw.

⇒ ORM allows us to develop objects based on DB slw independent persistence logic due to this o-r mapping persistence logical is portable across the multiple DB slw.

⇒ The process of mapping java class with DB table, java class properties with DB table columns and making java class objects representing DB table records having synchronization between them is called "o-r mapping."

⇒ Hibernate, IBatis, JDO, Toplink etc.. ORM slws.

⇒ Spring ORM is not direct ORM slw. It is spring module that provides abstraction layer on multiple ORM slw like hibernate, IBatis, JDO, JPA and etc.. and simplifies o-r mapping persistence logic development by providing Template classes to avoid boiler plate code problem

⇒ Spring ORM advantages

⇒ Common Transaction Management : Instead of performing each ORM slw specific programmatic Tx Management we can go for common spring style Declarative Tx Management.

⇒ Nested Tx Suppoit : Most of the ORM frameworks do not suppoite nested Txs directly. But spring suppoits Nested Txs. By using Spring ORM we can add nested Tx suppoit on o-r mapping persistence logic.

⇒ Portability : gives portability to change from 1 ORM slw to another ORM slw to develop persistence logic with minimum changes.

⇒ Avoids Boilerplate code : With the suppoit of multiple Template classes we can avoid boiler plate code (repeated code)

⇒ Common Exception Handling : Instead of handle each ORM specific checked or unchecked Exceptions Spring ORM throws them as common "Data Access Exception" heirarchy classes based exceptions and etc..

Note:

⇒ HibernateTemplate is designed based on Template design pattern.

Spring Hibernate Integration (Spring with Hibernate)

⇒ Spring with hibernate appln means developing hibernate style objects based OR mapping persistence logic by using hibernate template class in the DAO class of spring appln.

⇒ To create HibernateTemplate object we need HB sessionFactory object. We can use FactoryBeans like "LocalSessionFactory" or "AnnotationSessionFactory" classes as beans to create SessionFactory object by supplying hibernate cfg properties / cfg file and mapping file or mapping annotations based Domain class / HLO class and also a DataSource object.

⇒ This HibernateTemplate object will be injected to DAO classes to perform persistence operations.

HibernateTemplate's important methods

⇒ save(-), persist(-), update(-), delete(-), load(-) and get(-) for single row operations

⇒ bulkUpdate(-); To perform HQL based bulk non-select operations.

⇒ find(-) or findXxx(-) : To perform HQL/NativeSQL/criteria based select operations.

DB Table

SpEmployee
- ⊢→ eno (PK)
- ⊢→ ename
- ⊢→ esalary

ORMProg1

jars : IOC jars + Hibernate 3.6 jars + AOP jars + ojdbc14.jar + Apache DBCP jars + spring-orm-4.1.6.jar + spJDBC Lib.jars + AspectJ.jars + cglib.jar + aop-alliance.jar

- ⊢→ src
  - ⊢→ com.nt.cfgs
    - ⊢→ applicationContext.xml
    - ⊢→ service-beans.xml
    - ⊢→ persistence-beans.xml
    - ⊢→ aop-beans.xml
  - ⊢→ com.nt.domain
    - ⊢→ Employee(HLO).java
  - ⊢→ com.nt.dto
    - ⊢→ EmployeeDTO.java
  - ⊢→ com.nt.dao
    - ⊢→ EmployeeDAO.java, EmployeeDAOImpl.java
  - ⊢→ com.nt.service
    - ⊢→ EmployeeService.java, EmployeeServiceImpl.java
  - ⊢→ com.nt.controller
    - ⊢→ EmployeeController.java
- ⊢→ clientApp.java

Plain Hibernate Appln

⇒ Create Configuration object and call configure(-) method
⇒ Create Service Registry object by using ServiceRegistryBuilder
⇒ Create SessionFactory object
⇒ Open session with DB s/w
⇒ Begin Transaction
⇒ Perform persistence operation (app specific logic)

} (Common logics)

⇒ commit or rollback Tx
⇒ close session, SessionFactory object

} (Common logics)

In plain hibernate Appln we need to write both common and application specific logics.

Spring with Hibernate Appln (using spring ORM)

⇒ Inject HibernateTemplate object
⇒ Perform persistence operations

Executing HQL Query with Named Parameters

```
ht.findByNamedParam("from EmployeeHLO where esalary >=:min and esalary<=:max",
                    new String[]{"min","max"},
                    new Object[]{9000.0f, 10000.0f});
```

Executing HQL Query with Positional Parameters

```
ht.findByNamedParam("from EmployeeHLO where esalary >=:? and esalary<:?",
                    9000.0f, 10000.0f);
```

Note

⇒ While working with Hibernate we place HQL, NativeSQL in mapping file (or) Domain class as Named Queries. To make them visible in multiple session objects of one (or) more DAO classes.

Executing Named HQL Query with Positional Parameters

In EmployeeHLO.java (Domain class)

on top of Domain class

```
@NamedQuery(name="GET-EMPS-BY-SAL-RANGE", query="from EmployeeHLO
```

where esalary>=? and esalary<=? ")

In DAO class Method

List<EmployeeHLO> list = List<EmployeeHLO>
(ht.findByNamedQuery("GET_EMPS_BY_SAL_RANGE", 9000.of, 1000.of));

Executing NamedHQLQuery with NamedParameters

In Domain class (EmployeeHLO.java) → @NamedQuery(name="GET_EMPS_BY_SAL", query="from EmployeeHLO where esalary >=:min")

In DAO class Method List<EmployeeHLO> list = List<EmployeeHLO>
(ht.findByNamedQueryAndNamedParam("GET_EMPS_BY_SAL",
new String[]{"min"},
new Object[]{7000.of}));

⇒ If HibernateTemplate class does not provide environment to use certain feature of plain hibernate then take the support of HibernateCallBack Interface that executing gives doInHibernate(Session session)
* method having plain hibernate session object using that object we can finish of that operations.

Note For example,

⇒ We can't execute NativeSQL queries directly with HibernateTemplate class to make it possible we can use this HibernateCallBack Interface support.

In DAO class Method

List<EmployeeHLO> list = ht.execute(new HibernateCallback<List<EmployeeHLO>>(){
@Override
public List<EmployeeHLO> doInHibernate(Session ses) throws
HibernateException, SQLException{
//plain hibernate code
SQLQuery query = ses.createSQLQuery("select * from sp_Employee");
query.addEntity(EmployeeHLO.class);
List<EmployeeHLO> list = query.list();
return list;
}
});

ORMProj1

- src
  - com.nt.cfgs
    - hibernate.cfg.xml
    - persistence-beans.xml
  - com.nt.service
    - EmployeeService.java
  - com.nt.dao
    - Employee.java
    - EmployeeDAO.java
- test
  - ClientApp.java

hibernate.cfg.xml

```
<hibernate-configuration>
 <session-factory>
 <property name="connection.driver_class">oracle.jdbc.driver.OracleDriver</property>
 <property name="connection.url">jdbc:oracle:thin:@localhost:1521:xe</property>
 <property name="connection.password">scott</property>
 <property name="show-sql">true</property>
 <property name="connection.username">scott</property>
  <mapping class="com.nt.dao.Employee"/>
 </session-factory>
</hibernate-configuration>
```

```
<hibernate-configuration>
 <session-factory>
  <property name="connection.driver_class">com.mysql.jdbc.Driver</property>
  <property name="connection.url">jdbc:mysql:///NTDb1</property>
  <property name="connection.username">root</property>
  <property name="connection.password">root</property>
  <property name="dialect">org.hibernate.dialect.MySQL5Dialect</property>
  <property name="show-sql">true</property>
 <mapping class="com/nt/cfgs/Employee.hbm.xml"/>
 </session-factory>
</hibernate-configuration>
```

persistence-beans.xml

```
<bean id="dbcpds" class="org.apache.commons.dbcp.BasicDataSource">
<property name="driverClassName" value="oracle.jdbc.driver.OracleDriver"/>
<property name="url" value="jdbc:oracle:thin:@localhost:1521:xe"/>
<property name="username" value="scott"/>
<property name="password" value="tiger"/>
</bean>

<bean id="sesfact" class="org.springfm.orm.hibernate3.annotation.AnnotationSessionFactoryBean">
<property name="dataSource" ref="dbcpds"/>
<!--property name="configLocation" value="classpath:com/nt/cfgs/hibernate.cfg.xml"/-->
<property name="annotatedClasses">
<list>
<value>com.nt.dao.Employee</value>
</list>
</property>

<property name="hibernateProperties">
<props>
<prop key="show-sql">true</prop>
<prop key="dialect">org.hibernate.dialect.OracleLogDialect</prop>
</props>
</property>
</bean>

<bean id="template" class="org.sf.orm.hibernate3.HibernateTemplate">
<property name="sessionFactory" ref="sesfactory"/>
</bean>

<bean id="dao" class="com.nt.dao.EmployeeDAO">
<property name="ht" ref="template"/>
</bean>

<bean id="service" class="com.nt.service.EmployeeService">
<property name="dao" ref="dao"/>
</bean>
```

EmployeeDAO.java

```
public class EmployeeDAO{
    private HibernateTemplate ht;
    public void setHt(HibernateTemplate ht){
        this.ht = ht;
    }
    public int insert(Employee emp){
        int id = (Integer) ht.save(emp);
        return id;
    }
    public List<Employee> getEmpDetails(){
        List<Employee> list = ht.executeFind(new HibernateCallback(){
            @override
            public Object doInHibernate(Session ses) throws HibernateException, SQLExpti
                //write criteria logic
                Criteria ct = ses.createCriteria(Employee.class);
                Criteria cond1 = Restrictions.between("eno", 700,900);
                ct.add(cond1);
                List<Employee> list = ct.list();
                return list;
            }//method
        }//inner class
    };//method call
    return list;
    }// method
}
```

Employee.java

```
@Entity
@Table (name = "spEmployee")
//@NamedQuery (name ="test1 ", query = "from Employee where ename in (:name1, :name2)")
@NamedNativeQuery (name ="ntest1", query = "select * from spEmployee where eno>=:min
                                    and eno<=:max", resultClass = Employee.class)
public class Employee  implements java.io.Serializable {
        @Id
        @Column (name = "eno")
        private int eno;
        @Column ( name ="ename")
        private String ename;
       @Column (name = "esalary")
       private float esalary;
      public Employee() {
            S.O.P(" Employee : 0-param constructor");
       }
     public Employee (int eno, String ename, float esalary) {
        S.O.P (" Employee : 3-param constructor");
           this.eno = eno;
           this.ename = ename;
           this.esalary = esalary;
        }
   //setters and getters method
   @Override
  public String toString() {
     return "Employee [eno ="+eno +", ename =" +ename +", esalary="+esalary+ "]\n";
    }
}
```

EmployeeSevice.java

```
public class EmployeeSevice {
        private EmployeeDAO dao;
        private void setDAO (EmployeeDAO dao){
            this.dao = dao;
          }
    public String registerEmp (int no, String name, float salary) {
     //place given data in Employee class obj (Domain class obj)
     Employee emp = new Employee(no, name, salary);
       //call DAO method
      int id = dao.insert(Emp);
     return "Emp is registered with "+id;
    }
   public List<Employee> listAllEmployees(){
         List<Employee> list = dao.getEmpDetails();
           if(list != null)
             return list;
           else
             return null;
       }
  }
```

ClientApp.java

```
public class ClientApp {
      public static void main(String[] args){
  ClassPathXmlApplicationContext ctx = new ClassPathXmlApplicationContext ("/persistence-beans.xml");
   EmployeeSevice sevice = ctx.getBean ("sevice", EmployeeSevice.class);
     //call business method
    //S.O.P("Employee Registered? "+sevice.registerEmp (789, "Ramesh", 8000));
   S.O.P("All Emp Details " +sevice.listAllEmployees());
} }//main
```

5/10/15

Spring JEE/context Module

⇒ Provides abstraction layer on multiple JEE technologies and provides direct facilities to develop different kinds of Appln's.

⇒ This module provides

a) Application context container

b) Timer service for scheduling

c) Provides abstraction layer on RMI to develop spring RMI applns.

d) Provides abstraction layer on EJB to develop EJB client Appln.

e) Provides abstraction layer on JMS to develop spring JMS application for messages based on asynchronous communication

f) Gives Httpinvoker, Burlap and etc.. as direct distributed technologies to develop Distributed Appln

scheduling

⇒ keeping certain task to execute automatically at regular intervals (or) at certain time is called "scheduling". It is all about enabling timer service logics that we want to execute.

⇒ While dealing with remainder Applns, passing data for marketing from 1 place to another place we need scheduling support.

⇒ Youtube, facebook operational data → Broker agency → Dealers → uses that for marketing.
(1) (Categories the data) (2) (3)

(1), (2), (3) → Develops the appln to execute based on scheduling at regular intervals, to pass the data every day.

⇒ While developing time critical jobs we need scheduling ..... taking backup, doing date, time adjustments and etc...

⇒ In jdk Timer, TimeTask classes are given for scheduling .....

⇒ TimeTask → To define job on which we want to enable scheduling

⇒ Timer → To specify scheduling.
Here we specify initialDelay, period values
for example, ↦ The time gap between two end to end backups to back

Refer page NO: 62 of MVC.

Sample code for jdk based scheduling appln.

Spring Based Timer/Scheduling Appln

⇒ Allows to enable scheduling on multiple jobs at a time.

⇒ Allows to specify jobs in user-defined methods of user-defined classes and we can specify initial delay, period params through xml cfgs declaratively.

⇒ ScheduledTimerTask: Allows to hold the job that is defined in the subclass of TimerTask class and can also specify initialDelay, period params

⇒ MethodInvokingTimeTaskfactoryBean: Returns TimerTask class object having job defined in user-defined method (not the run method).

⇒ TimerFactoryBean: Starts TimerService on the startup of ApplicationContext container and stops the TimerService once it is down.

Refer page NO: 62 & 64 of MVC.

6/10/15

Spring Security

⇒To perform dependency injection on servlet filter component its class must be maintained by IOC container as spring bean having bean configuration in spring bean configuration file, Due to this it can not take request from client as filter is not configured as web component in web.xml. To overcome this problem we configure 1 special Proxy filter in web.xml to take request from client to pass the request to filter component that is there in IOC container as spring bean, the special filter is "org.sf.web.filter.DelegatingProxyFilter". But we need to match logical name (given in <file-name>) of DelegatingProxyFilter with bean name of filter component managed by IOC container.

Security (Authentication + Authorization)

↦ checking the identity of user is called "Authentication" (verifying username, password)

↦ checking the access permissions of the user to utilize the services is called "Authorization".

⇒ Provide authorization or access permission not based on usernames... provide them based on roles. Roles are like designations.

⇒ checking the identity of users to use banking appln is called "authentication."

⇒ checking the access permissions of a user based his roles to use various services is called "authorization".

While implementing security we need the following services

a) Authentication provides / Authentication info provider

⇒ It is the place where we can maintain set of username and passwords that are required for authentication. we can take

a) File system file (Text file, xml file and etc..)

b) DB slw

c) LDAP server (Light Weight Directory Access Protocol)

⇒ LDAP is dedicatedly given to maintain usernames and passwords. The specially once forget the password we can not get password but we can reset the password.

⇒ It is also to maintain username, password that required in single signon operations (login once use them for multiple Applns)

Eg: Logon to gmail and use in youtube.com, google drive, google + and etc..

b) Authentication Manager

⇒ It is seperate component that is linked with Sewlet Filter to verify to given username, password with the username, password that are available in Authentication provider. If valid allows the request going to the actually requested resources, If not valid shows 401, 403 error pages.

⇒ This has to ignore authentication when request is given some resources like home page and has to enable authentication for other resources.

⇒ In Non spring environment we can implement security in two ways.

a) Programmatic / Custom Security Model

⇒ Here we need to write code of Authentication Manager takes requests, render dialog boxes using user names, passwords, talks to Authentication provider for verification and either allows requests to go to requested resource or blocks the request by throwing 404 or 403 Error page. Doing this work manually it is very complex one. so it is not recommanded Model.

b) Sewer managed Declarative Security Model.

⇒ Here Authentication Manager work, Authentication provider will be taken care by Sewer/container based on the configurations done in web.xml file to reduce the burden on programmers but still some limitations are there in this approach.

a) web.xml entries are not protal across the multiple sewers.
b) Sewer webSewers do not support this model
c) Going to commerical App sewer on for security model is cost effective.
d) Some sewer do not support LDAP sewer as authentication provider.

⇒ To overcome this problem use spring security Enable Model earlier name is Springacegi security

a) It runs in all sewers and containers irrespective of they support security or not
b) Supports all model authentication bean
c) It also supports Declarative Security Model
d) Can be applied on normally web application and spring Mvc web applications and etc..

Note We can spring security definitions in spring bean cfg file having security namespace, based on these cfgs internally one security SewletFilter will be generated as spring Bean of IOC container having bean name: "springSecurityFilterchain" so to take requests from client and to pass to requests to the above generated filter we need to configure "DelegatingProxyFilter" in web.xml file with logical name "springsecurity Filterchain".