Framework:

1. It is a special software that is capable of developing applications based on ceratian architecture with the ability to generate common logics of the application.
2. It is given based on core technologies.
3. It provides an abstraction layer on core technologies.

Struts: It is a framework to develop MVC-II architecture based web applications only.
Hibernate: Its a ORM Framework which is used only for database operations.
Spring: Its a framework software which can be used to develop any kind of Java/J2EE applications.
Note: Spring framework provides abstraction layer on:

1. java & J2EE core technologies
2. ORM Tools like:
    1. Hibernate
    2. JDO
    3. iBastis etc..,
3. AOP Framework etc..,

Spring Info:

Type: java-j2ee Framework software
Version: 2.5[Compatible with jdk1.5+]
Vendor: Interface21
Creator: Rod Johnson
Open Source
Download: www.springframework.org
Online Tutorials: www.roseindia.net
Articles:

1. www.javabeat.net
2. www.onjava.com
3. www.precisejava.com
4. www.devx.com

FAQ's: www.forum.springframework.org
Reference Books:

1. Spring Live
2. Spring in Action --> Manning Publishers

Introduction:

1. Spring framework is developed to simplify the development of enterprise applications in Java technologies.
2. It is an open source framework begin developed by Interface21.
3. The Spring provides light weight IoC Container and AOP framework.
4. It provides support for JPA, Hibernate, Web services, Schedulers, Ajax, Struts, JSF and many other frameworks.
5. The Spring MVC components can be used to develop MVC based web applications.
6. The Spring 3.0 framework has been released with major enhancements and support for the Java 5 [JDK1.5].
7. Spring can be used to configure declarative transaction management, remote access to your logic using RMI or web services, mailing facilities and various options in persisting your data to a database.
8. Spring framework can be used in modular fashion, it allows to use in parts and leave the other components which is not required by the application.

Spring Architecture: Spring is well-organized architecture consisting of seven modules. Modules in the Spring framework are:

Spring1.x: It has 7 modules. They are:

1. Core
2. DAO
3. ORM
4. JEE/Context
5. Web
6. MVC
7. AOP

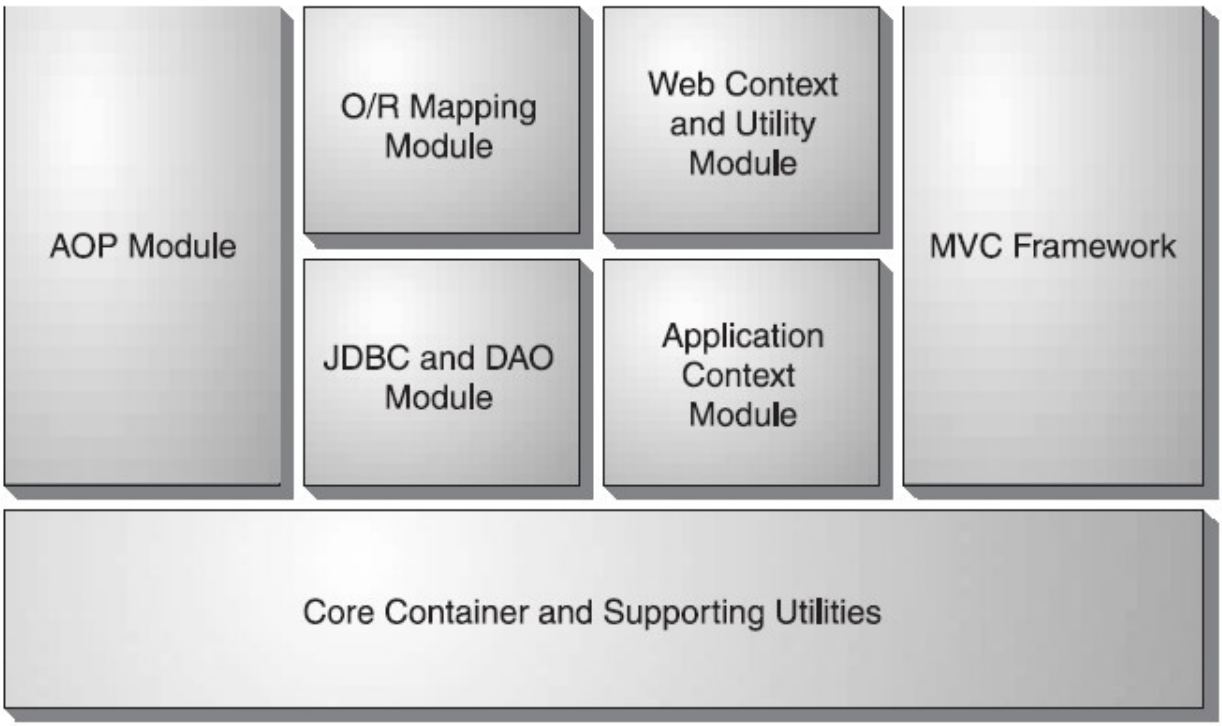


Figure: The Spring framework is composed of several well-defined modules.

Spring2.x: It has 6 modules only. Because Web & Web MVC modules were clubbed together & given as Web MVC module.

Core Module:

1. It is base module for all modules.
2. Provides BeanFactory container which makes Spring as a container
3. Acts as a base module for other modules
4. provides the fundamental functionality of the Spring framework.
5. The Core package is the most import component of the Spring Framework.
6. This component provides the Dependency Injection [DI] features.

DAO Module:

1. It provides abstraction layer on JDBC & allows us to do database operations.
2. Using it we can focus only on database operations instead of doing repeated work by writing common code again and again.

3. Here we can make use of JdbcTemplate class where we can call methods to do required database opeartion.
4. We can connect to database either by manual code same like jdbc or we can do easily by using JdbcTemplate class.
5. Using JdbcTemplate class we can retrieve data from database in the form of serializable objects.

ORM Module:

1. It provides an abstraction layer on ORM Framework software's like Hibernate, iBatis, JDO etc..,
2. Spring doesn't attempt to implement its own ORM solution
3. Spring's transaction management supports each of these ORM frameworks as well as JDBC.

JEE/Application Context Module:

1. It provides abstraction layer on Java/J2EE technologies like Java Mail, JMS, EJB etc..,
2. The context module is what makes spring as a framework.
3. This module extends the concept of BeanFactory
4. Adds support for internationalization (I18N) messages, application life cycle events, and validation.
5. Supplies many enterprise services such as e-mail, JNDI access, EJB integration, remoting, and scheduling.

AOP [Aspect Oriented Programming] Module:

1. One of the key components of Spring is the AOP framework.
2. It provides a methodology to configure middleware services or to perform any authentication, validations or to perform any pre processing logic or post processing logic on our business logic methods.
3. Without disturbing any kind of spring resources we can integrate our spring application with any kind of middleware service.
4. Spring provides a separate layer for middleware configuration.
5. This module serves as the basis for developing your own aspects for your Spring-enabled application.

Web MVC Module: It is given for two operations:

1. To integrate spring application with any other web based framework like Struts, JSF etc..,
2. It provides its own web framework anmed as Spring MVC to develop MVC-II architecture based web applications. It provides the MVC implementations for the web applications.

Note: We can develop a spring application either by using all modules or only by using few modules of spring. We can also integrate our spring module with any kind of ajava application. So spring is **loosely coupled**.

Spring is a open source, light weight, loosely coupled, aspect oriented & dependency injection based framework software which can be used to develop any kind of java/j2ee application.

Open Source: Spring software can be downloaded and used at free of cost. Spring developed source code can referred by any one who is interested.

Container is a special software which can manage complete life cycle of a given resource.

Loosely Coupled: Need not to inherit any class or interface as we do for our servlet programming. Instead of having "is-a" relationship here we can have "has-a" relationship.

Lightweight:

1. Spring is lightweight in terms of both size and overhead.
2. The entire Spring framework can be distributed in a single JAR file that weighs in at just over 1 MB.
3. **Spring is nonintrusive:** objects in a Spring-enabled application typically have no dependencies on Spring specific classes.
4. Mainly spring is used for business logic. The same business logic if we try to develop using EJB, then its compulsory to have support of any application server like weblogic. But the same logic can be executed in Spring without help of any web server or application server. Because Spring provides its own containers in the form of pre defined classes.

Note: Servlet Container & EJB Container are heavy weight because unless we start web server or application server we cannot activate either servlet container or EJB Container.

Inversion of control:

1. Spring promotes loose coupling through a technique known as inversion of control (IoC).
2. When IoC is applied, objects are passively given their dependencies instead of creating or looking for dependent objects for themselves.
3. The container gives the dependencies to the object at instantiation without waiting to be asked.

Aspect-oriented:

1. Spring comes with rich support for aspect-oriented programming.
2. It enables cohesive development by separating application business logic from system services (such as auditing and transaction management).
3. Application objects do what they're supposed to do—perform business logic—and nothing more. They are not responsible for (or even aware of) other system concerns, such as logging or transactional support.

Container*:*

1. Spring is a container in the sense that it contains and manages the life cycle and configuration of application objects.
2. We can configure how each of our beans should be created—either create one single instance of bean or produce a new instance every time one is needed based on a configurable prototype.

Enterprise Application: Its a large scale appliaiton with complex business logic & configured with middleware services like transaction management, security etc.., Eg: Banking Applications.

Dependency Lookup: This concept is used to find required resource among multiple resources. It takes some time to look up waht we need. It performs "PULL" operation to get required values.
Eg: Getting JDBC DataSource object from registry server.
Note: Registry servers are used to provide global visibility [to access from any where through out the world].
Eg: RMI Registry

Dependency Injection [IOC – Inversion Of Control]: Container software will gather all the required values and injects into resources of application dynamically. Here Dependent values will be pushed to resources dynamically by underlying container.
Ex: The way how JVM/JRE calls constgructor of a class to assign some values at the time of object creation.

Note: Using spring we can develop any kind of Java/J2EE application. But mostly in industry we can find spring used for development of Businesss Logic.

Installation of spring: Download software & extract "**Spring-framework-2.5.1-with dependencies.zip**" file.

Spring Jars: To work with any kind of spring applicaiton we need to set the classpath for two jar files. They are:

1. spring.jar [main jar file] [**Location: sping-home\dist\spring.jar**]
2. **commons-logging.jar [Dependency jar file] [Location:]**

Normal Spring Application:

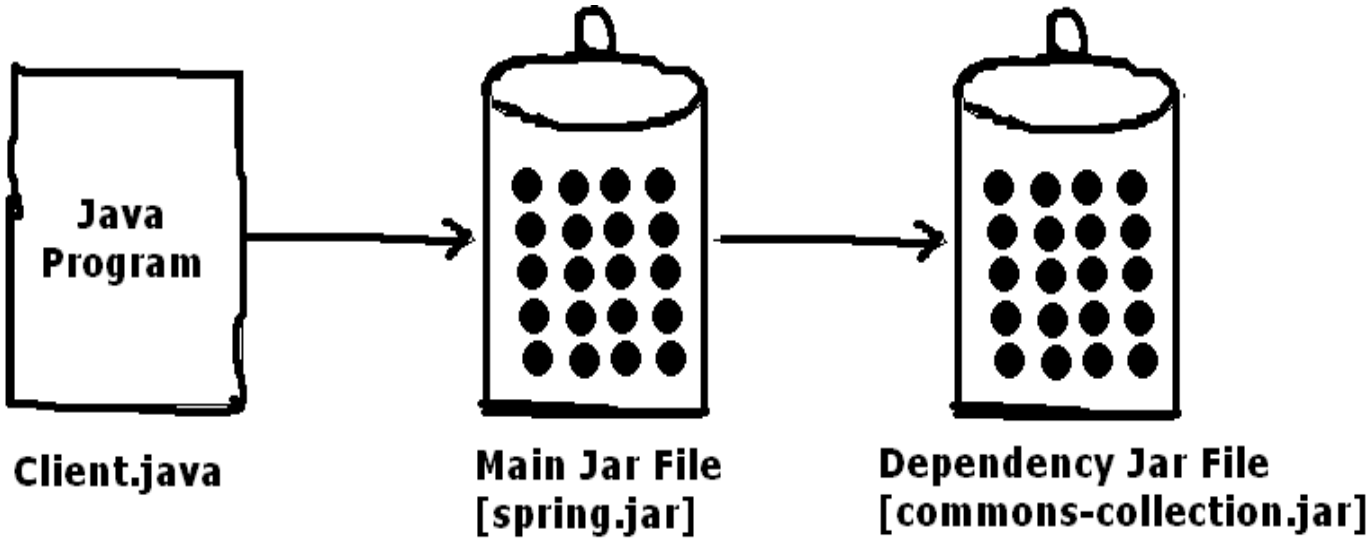
1. **Compilation:** Set the classpath for main jar file
2. **Execution:** Set the classpath for both main & dependency jar file. Becuase at run time of program the control transfers from main jar file to dependency jar file.

Spring Web Applicaiton:

1. **Compilation:** Set the classpath for main jar file
2. **Execution:** Place both main & dependency jar files inside "lib" folder of WEB-INF.

Note: Spring programs will make use of classes and interfaces resideing inside "spring.jar" file and the classes which are residing inside "spring.jar" file can make use of classes & interfaces which resides in dependency jar file "commons-logging.jar".

Classpath: So when ever we need to work with a spring applicaiton its compulsory to set the classpath for all related main jar files and also dependent jar files.



Process of classpath setting: My Computer icon -> advanced -> environment variables -> system variables -> [if classpath variable is existing -> edit, if classpath variable doesn't exist then create a new classpath variable] & specify c;asspath.

Note:

1. When we need set the classpath for multiple jar files, every jar file need to be specified with absolute path and every jar file path should be separated with semiclon [**;**]
2. The classpath what we specify in environmental variables will not be reflected to any ide like netbeans or eclipse or myeclipse etc..,
3. When we configure the same variable in user variable & system variable then user variables will dominate system variable [for path or classpath]
4. If any jar file missed in middle of classpath then the rest of classpath will be ignored. So always we need to specify our classpath in beginning of the path.

Features of Spring:

1. Supports POJO/POJI model programming.
2. Provides two light weight containers in the form of pre defined classes.
    1. **Bean Factory Container** or **Core Container** or **IOC Container**
       -org.springframework.beans.factory.BeanFactory Interface
    2. **JEE Container** or **Application Context Container.**
       -org.springframework.beans.factory.xml.ApplicationContext interface
3. Spring containers are activated from server side programs.
4. Can be used to develop any kind of java/j2ee applications
5. We can develop an applicaition only by sing spring or we can also integrate spring with any other framework.
6. Provides built in middleware services like connection pooling, transaction management, logging etc.,
7. Supports to work with third party supplied and web server or application server supplied middleware services.
8. Supports nested transaction & distributed transaction management.
9. Supports AOP framework to integrate with middleware services.
10. Provides anootation based mapping as a alternative to xml configuration which increases performance of the application.
11. **Transaction Management:**
    1. Spring framework provides a generic abstraction layer for transaction management.
    2. This allows the developer to add the pluggable transaction managers, and making it easy to do transactions without dealing with low-level issues.
    3. Spring's transaction support is not tied to J2EE environments and it can be also used in container less environments.
12. **JDBC Exception Handling:** The JDBC abstraction layer of the Spring offers a meaningful exception hierarchy, which simplifies the error handling strategy

Specific API: API which is used for development of only a particular kind of application. Eg: servlet-api is used only for the purpose of web application development.

POJO Classes: The classes that are not API dependent are called as POJO [Plain Old Java Object] classes.

POJO: [plain old java object] A pojo class is a class which should not extend any predefined class belongs to specific API and it should not implement any predefined interface belongs to specific api.

Following are pojo classes:

```
class C1
{
}
class C2 extends C1
{
}
class C3 extends java.lang.Thread
{
}
```

Note: Thread is a predefined class but its not confined for development of any particular kind of application. In another words Thread class can be used for any kind of java application. So our class C3 is a pojo class.

```
class C4 implements java.lang.Runnable
{
}
```

Note: Runnable is an interface which can be used to create a thread. So it can be used for development of any kind of java application.

```
class C5 implements java.io.Serializable
{
}
```

Note: Serializable interface implementation class objects will become serializable objects. So that we can transfer from one location to another location over the network.

Following are not POJO classes:

```
class C6 extends GenericServlet
{
}
```

Note: GenericServlet class is used only for the purpose of web application development. So our class is not a pojo class.

```
class C7 implements Servlet
{
}
```

Note: Our class is implementing Servlet interface which is confined for development of web applications.

```
class C8 implements java.rmi.Remote
{
}
```

Note: Remote interface is used only for development of Distributed Applications. So the above class is not a pojo class.

POJI Interfaces: The interfaces that are not API dependent are called as POJI [Plain Old Java Interface]

POJI: [Plain Old Java Interface] Poji is an interface which should not extend any predefined interface which belongs to specific API.

Examples of POJI are:

```
interface I1
{
}
interface I2 extends I1
{
}
interface I3 extends Runnable
{
}
interface I4 extends Serializable
{
}
```

Examples of non POJI are:

```
interface I5 extends Remote
{
}
interface I6 extends Servlet
{
}
```

Note: The above interfaces are non poji because they are extending interfaces which are api specific.

Server: A program which is providing service is called as server.

Client: A program which is giving request to get some service is called as client program.

Local Client: If server and client both are executing under same JVM [Java Virtual Machine] then that client program can be called as a Local Client and server can be called as Local Server.

Remote Client: If server and client both are executing under different JVM's then we can consider client as Remote client to server and server as Remote Server to client program.

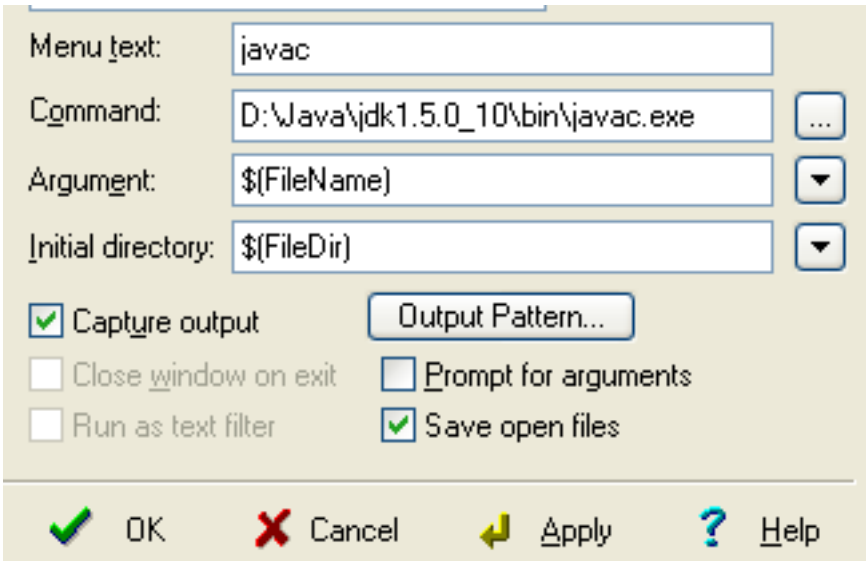
Spring Containers:

1. **Core container or IOC container or BeanFactroy container**
    1. Its an implementation class of an interface BeanFactory
    2. It supports **lazy loading**.
        1. It means container will not create objects of cofigured pojo classes immediately.
        2. It waits till it recieves the first request.
        3. Once the object is created the same object will be maintained till end of the application.
    3. By default it considers all the configured classes as singleton classes.
    4. So that the same object will be given to all requests which belongs to same class.
    5. We cannot change singleton scope for core container.
    6. provides the fundamental functionality of the Spring framework.
    7. BeanFactory, the heart of any Spring-based application.
2. **Advanced container or ApplicationContext container**
    1. It is advanced container.
    2. By default it supports early loading.
        1. It means without recieving any request from client program immediately it will create object for every singleton scoped class.

2. When ever the container gets acivated immediately it creates objects of all singleton classes.
3. Has many advantages over core container.

Execution of java program using Edit Plus editor: Once we configure Edit Plus we can compile and execute java programs just by using short cut keys. So it provides a great flexibility to develop java programs.

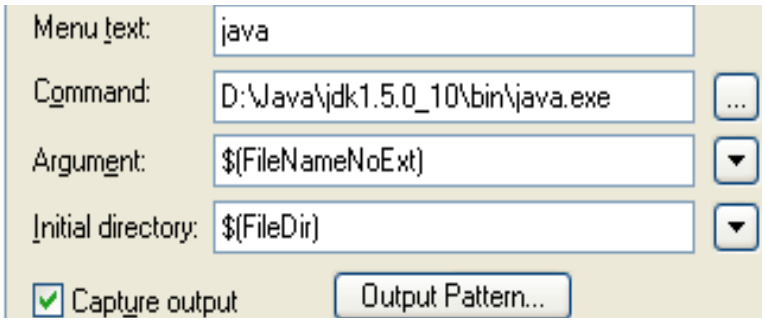
Configure Compilation process: Tools [menu bar] → configure user tools → add tool → program → give menu text as javac/compile [which ever is comfortable to you] → for command click on button with 3 dots [browse button] and locate **javac** from **bin** folder of JDK software.



Argument → File Name →
Initial Directory → File Directory
Select check box of capture output → Apply → ok.

Configure Execution process:

→ give menu text as java/Execute [which ever is comfortable to you] → for command click on button with 3 dots [browse button] and locate **java** from **bin** folder of JDK software.



Argument → File Name Without Extension →
Initial Directory → File Directory
Select check box of capture output → Apply → ok.

Note: Click on tools it shows you short cut keys to execute our java programs.

Resources of spring core module application:

1. **Interface [POJI]**
    1. It is optional in all applications except distributed applications.
    2. It can contain prototype of our business logic methods.
2. **Spring Bean class [POJO class]**
    1. It can be implementation class of our interfacc
    2. It contains definition for business logic methods
    3. It can have properties and values can be injected by spring container through DI [Dependency Injection].
    4. It can have setters methods or constructors to support dependency injection.
    5. It can have user defined methods which can act as life cycle methods.
    6. Along with business logic methods we can also have helper methods.

3. **Spring Configuration file [*.xml file]**
    1. We can give any name for this xml file
    2. standard name is to give as "applicationContext.xml"
    3. Initially we need to specify either **DTD (Document Type Definition)** or **XSD (XML Schema Definition)**
    4. Xml file can be validated based up on either XSD or DTD.
    5. Its compulsory to specify either DTD or XSD. [DTD is traditional & XSD is advanced]
    6. We can configure all bean classes [business logic classes] here.
    7. We can configure user dfined classes & also predefined classes.
    8. Every class can have a unique identification name.
    9. We can also configure bean class properties here. Through this we can pass instructions to spring container to inject values with dependency injection.
    10. Spring container works based up on the instructions what we have given in spring configuration file.
4. **Client Program [*.java]**
    1. Locates spring configuration file
        1. To locate spring configuration file regulary we will be using either **ClasPathResource** or **FileSystemResource**
        2. **Resource** is interface
        3. **ClasPathResource** & **FileSystemResource** both are implementation classes or **Resource** interface.
        4. If client program & spring configuration file both resides in same folder then we can use any implementation class without any problem.
        5. **FileSystemResource:**
            1. If it is required to specify either **abosolute path [Eg: c:\folder-name\file-name.xml]** or **relative path [Eg: sub-folder-name\file-name.xml]** in those cases we need to use **FileSystemResource** class.
            2. **Drawback:** If we modify location of spring configuration file then again we need to modify either aboslute path or relative path in our program.
        6. **ClasPathResource:**
            1. We can create a jar file containing our spring configuration file and we can place it in our classpath.
            2. So that the spring configuration file can be any where. But still our client program can access it.
            3. **Drawback:** If we change our spring configuration file then again we need to create a jar file freshly.
    2. Activates spring container

1. Here we can use **BeanFactory** interface and allocate memory of **XmlBeanFactory** class which is an implementation class of BeanFactory interface.
3. **Reads spring configuration file:** Once the container is activated automatically it reads spring configuration file.
4. Requests spring container to give object of any bean class by using identification name of the bean class.
5. By using bean class object we can call business logic methods.

Example: Simple Spring Application.

//interface program

//DemoInter.java

```
public interface  DemoInter
{
        public String wish(String uname);
}
```

//Implementation class

//DemoInterImpl.java

```
public class  DemoInterImpl  implements DemoInter
{
        private String  message;

        public DemoInterImpl()
        {
                System.out.println("Constructor executed");
        }

         public void setMessage(String message)
        {
                System.out.println("setter method executed");
                this.message=message;
        }

        public String wish(String uname)
        {
                return   message+" "+uname+" Have a great day";
        }
}
```

Note: We can copy paste DTD or XSD int ospring configuration file by searching xml files provided by our spring software.

Spring Configuration file

//spring.cfg.xml

//Here we are specifying DTD (Document Type Definition)

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC  "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
```

```
<beans>
    <bean id = "ic" class="DemoInterImpl">
            <property  name="message">
                    <value>Hello</value>
            </property>
    </bean>
</beans>
```

(or)

//Here we are specifying XSD (XML Schema Definition)

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">

          <bean id = "ic" class="DemoInterImpl">
               <property  name="message">
                       <value>Hello</value>
               </property>
          </bean>
</beans>
```

//Client Program
//DemoClient.java

```
import org.springframework.core.io.Resource;
import org.springframework.core.io.ClassPathResource;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
public class   DemoClient
{
      public static void main(String args[])
      {
         //location of spring configuration file
         Resource  res=new ClassPathResource("spring.cfg.xml");

         //activation of spring container & then it reads spring configuration file
         BeanFactory  factory=new XmlBeanFactory(res);

         //requesting object of a bean class by using its identification name
         DemoInter d=(DemoInter)factory.getBean("ic");

         //calling business logic method
         System.out.println(d.wish("Sai"));
      }
}
```

Note: To execute any spring program basically we need to have the following jar files in classpath.

Main Jar file: **spring-framework-2.5.1\dist\spring.jar**

Dependency Jar file:
spring-framework-2.5.1\lib\jakarta-commons\commons-logging.jar

Singleton Demo:
Note: All resources are same as above program. Only change in client program.

```
import org.springframework.core.io.Resource;
import org.springframework.core.io.ClassPathResource;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
public class   DemoClient
{
      public static void main(String args[])
      {

         Resource  res=new ClassPathResource("Demo.xml");
         BeanFactory  factory=new XmlBeanFactory(res);

         DemoInter d1=(DemoInter)factory.getBean("ic");
         DemoInterImpl d2=(DemoInterImpl)factory.getBean("ic");
         DemoInter d3=(DemoInter)factory.getBean("ic");

         System.out.println(d1.wish("Sai"));
      }
}
```

Note:

1. getBean() is a method which returns us super class **Object.** So we can classcast it to either our interface or implementation class.
2. We have requested the same class object for 3 times. But container creates only one object & the3 same object will be given for every request.
3. We can have any number of business logic methods inside the same class.

Example: Configuring a predefined class object as property of pojo class.
//spring.cfg.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC  "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">

<beans>

        <bean  id="dat"   class="java.util.Date">
              <property  name="year">
                    <value>85</value>
              </property>
```

```
502                 <property  name="month">
503                         <value>10</value>
504                 </property>

505                 <property  name="date">
506                         <value>19</value>
507                 </property>
508             </bean>

509     <bean  id="demo"   class="DemoInterImpl" >
510                 <property  name="message">
511                         <value>Njoy Spring Programming :)</value>
512                 </property>

513                 <property  name="dt" >
514                         <ref bean = "dat"/>
515                 </property>
516     </bean>
517 </beans>

518 //DemoInter.java
519 public interface  DemoInter
520 {
521         public void show();
522 }

523 //DemoInterImpl.java
524 import java.util.Date;

525 public class  DemoInterImpl  implements DemoInter
526 {
527         private String  message;
528         private Date dt = null;

529         public DemoInterImpl()
530         {
531                 System.out.println("DemoInterImpl zero argument constructor");
532         }

533         public void setMessage(String s)
534         {
535                 System.out.println("setMessage() of DemoInterImpl class");
536                 message=s;
537         }

538         public void setDt(Date dt)
539         {
540                 System.out.println("setDt() of DemoInterImpl class");
541                 this.dt = dt;
542         }
```

```
543         public void show()
544         {
545                 System.out.println("Mesage is: "+message);
546                 System.out.println("Date is: "+dt);
547         }
548 }
```

Client.java

```
549 import org.springframework.core.io.Resource;
550 import org.springframework.core.io.ClassPathResource;
551 import org.springframework.beans.factory.BeanFactory;
552 import org.springframework.beans.factory.xml.XmlBeanFactory;
553 public class   DemoClient
554 {
555       public static void main(String args[])
556       {
557                       Resource  res=new ClassPathResource("spring.cfg.xml");
558                       BeanFactory  factory=new XmlBeanFactory(res);
559                       DemoInter d1=(DemoInter)factory.getBean("demo");
560                       d1.show();
561       }
562 }
```

//Example: Configuring a user defined class object as property of pojo class.

//DemoInter.java

```
563 public interface  DemoInter
564 {
565        public void show();
566 }
```

DemoInterImpl.java

```
567 import java.util.Date;
568 public class  DemoInterImpl  implements DemoInter
569 {
570         private int age;
571         private Date dt = null;  //pre defined class
572         private TestBean tb = null; //user defined class
573         public DemoInterImpl()
574         {
575                 System.out.println("DemoInterImpl zero argument constructor");
576         }
577         public void setAge(int age)
578         {
579                 this.age = age;
580         }
581         public void setDt(Date dt)
582         {
583                 this.dt = dt;
584         }
```

```
589         public void setTb(TestBean tb)
590         {
591                 this.tb = tb;
592         }
593         public void show()
594         {
595                 System.out.println("Age is: "+age);
596                 System.out.println("Date is: "+dt);
597                 System.out.println("Message is: "+tb);
598         }
599 }

600 //spring.cfg.xml
601 <?xml version="1.0" encoding="UTF-8"?>
602 <!DOCTYPE beans PUBLIC  "-//SPRING//DTD BEAN//EN"
603 "http://www.springframework.org/dtd/spring-beans.dtd">
604 <beans>
605          <bean  id="dat"   class="java.util.Date">
606                 <property  name="year">
607                         <value>85</value>
608                 </property>
609                 <property  name="month">
610                         <value>10</value>
611                 </property>
612                 <property  name="date">
613                         <value>19</value>
614                 </property>
615           </bean>
616          <bean  id="tbean"   class="TestBean" >
617                 <property  name="message">
618                         <value>Hello friends.... Njoy this beautiful n gifted life :)</value>
619                 </property>
620         </bean>
621          <bean  id="demo"  class="DemoInterImpl">
622                 <property  name="age">
623                         <value>25</value>
624                 </property>
625                 <property  name="dt">
626                         <ref bean = "dat"/>
627                 </property>
628                 <property  name="tb" ref = "tbean"/>
629                 <!--     <ref bean = "tbean"/>
630                 </property> -->
631         </bean>
632 </beans>
633 //DemoClient.java
634 public class   DemoClient
635 {
636       public static void main(String args[])
637       {
```

```
                    Resource  res=new ClassPathResource("spring.cfg.xml");
                    BeanFactory  factory=new XmlBeanFactory(res);
                    DemoInter d1=(DemoInter)factory.getBean("demo");
                    d1.show();
        }
}
```

//Example: Different Bean Property Types

Note: In a bean class we can have any type of properties. For the following properties we have corresponding tags to be onfigured in our spring configuration file.

Bean Property type	**Tag in spring configuration file**
Primitive (or) java.lang.String	<value>
Other bean type [class/interface type]	<ref>
Java.util.List type [stack, linkedlist]	<list>
Java.util.Set	<set>
Java.util.Map	<map>
Java.util.Property	<props>
Java.util.Array or arrays	<list>

//DemoInter.java

```
public interface DemoInter {
        public String sayHello();
}
```

//DemoImple.java

```
import java.util.Date;
import java.util.List;
import java.util.Map;
import java.util.Properties;
import java.util.Set;
public class DemoImple implements DemoInter{
        private float salary;
        private TestBean tb = null;
        private Date dt = null;
        private List fruits = null;
        private List veg = null;
        private Set phone_nos = null;
        private Set emails = null;
        private Map perDetails = null;
        private Map capitals = null;
        private Properties faculties = null;
        private String courses[];
        private int runs[];
        public String sayHello() {
                return "Hello Friends... Gmg. :) "+
                " salary = "+salary+
                " tb = "+tb.toString()+
                " dt = "+dt.toString()+
                " fruits = "+fruits.toString()+
                " veg = "+veg.toString()+
```

```
677                 " phones = "+phone_nos.toString()+
678                 " emails = "+emails.toString()+
679                 " perDetails = "+perDetails.toString()+
680                 " capitals = "+capitals.toString()+
681                 " faculties = "+faculties.toString()+
682                 " courses = {"+courses[0]+" "+courses[1]+"}"+
683                 " runs = {" + runs[0]+" "+runs[1]+"}";
684         }
685         public void setSalary(float salary) {
686                 this.salary = salary;
687         }
688         public void setTb(TestBean tb) {
689                 this.tb = tb;
690         }
691         public void setDt(Date dt) {
692                 this.dt = dt;
693         }
694         public void setFruits(List fruits) {
695                 this.fruits = fruits;
696         }
697         public void setVeg(List veg) {
698                 this.veg = veg;
699         }
700         public void setPhone_nos(Set phone_nos) {
701                 this.phone_nos = phone_nos;
702         }
703         public void setEmails(Set emails) {
704                 this.emails = emails;
705         }
706         public void setPerDetails(Map perDetails) {
707                 this.perDetails = perDetails;
708         }
709         public void setCapitals(Map capitals) {
710                 this.capitals = capitals;
711         }
712         public void setFaculties(Properties faculties) {
713                 this.faculties = faculties;
714         }
715         public void setCourses(String[] courses) {
716                 this.courses = courses;
717         }
718         public void setRuns(int[] runs) {
719                 this.runs = runs;
720         }
721 }
```

722 **//Note: Set doesn't allows duplicates but List allows**

//spring.cfg.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans
	xmlns="http://www.springframework.org/schema/beans"
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://www.springframework.org/schema/beans
	http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">
	<bean id ="dat" class = "java.util.Date"/>
	<bean id ="tbean" class = "TestBean">
		<property name="msg">
			<value>Hello</value>
		</property>
	</bean>
	<bean id = "db" class = "DemoImple">
		<property name="salary">
			<value>5000</value>
		</property>
		<property name="tb">
			<ref bean = "tbean"/>
		</property>
		<property name="dt">
			<ref bean = "dat" />
		</property>
		<property name="fruits">
			<list>
				<value>Apple</value>
				<value>Banana</value>
				<value>Orange</value>
			</list>
		</property>
		<property name="veg">
		<list>
			<value>Carrot</value>
			<value>Potato</value>
			<value>Mushrooms</value>
			<!--  any objects can be stored in a List-->
			<ref bean = "dat"/>
			<ref bean = "tbean"/>
			<value>Tomato</value>
			</list>
		</property>
		<property name="phone_nos">
			<set>
				<value>7569670669</value>
				<value>9502166767</value>
			</set>
		</property>
```

```
770             <property name="emails">
771                 <set>
772                     <value>kanakavaraprasad@gmail.com</value>
773                     <value>s.varaprasad@yahoo.co.in</value>
774                 </set>
775             </property>
776             <property name="perDetails">
777                 <map>
778                     <entry>
779                         <key>
780                             <value>Sai</value>
781                         </key>

782                         <value>Universal King</value>
783                     </entry>

784                     <entry>
785                         <key>
786                             <value>Kanakadhar</value>
787                         </key>
788                         <value>Service Provider</value>
789                     </entry>
790                 </map>
791             </property>
792             <property name="capitals">
793                 <map>
794                     <entry>
795                         <key>
796                             <value>Sathya</value>
797                         </key>

798                         <value>Boon for learners</value>
799                     </entry>

800                     <entry>
801                         <key>
802                             <ref bean = "dat"/>
803                         </key>

804                         <ref bean = "tbean"/>
805                     </entry>
806                 </map>
807             </property>
808             <property name="faculties">
809                 <props>
810                     <prop key="Kanakadhar">Java Faculty</prop>
811                     <prop key="Mahesh">.net Faculty</prop>
812                 </props>
813             </property>
```

```
<!--Following both properties courses & runs are array variables. So we made use of list tag--
>
        <property name="courses">
            <list>
                <value>Java</value>
                <value>.net</value>
                <value>Oracle</value>
            </list>
        </property>

        <property name="runs">
            <list>
                <value>30</value>
                <value>40</value>
            </list>
        </property>
    </bean>
</beans>
```

Note: While configuring dependent values for collection framework type bean properties, we can configure one data structure object as the element value of another data structure.

//TestBean .java

```
public class TestBean {
    String msg;

    public void setMsg(String msg) {
        this.msg = msg;
    }
    public String toString()
    {
        return msg;
    }
}
```

//DemoClient.java

```
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;

public class DemoClient {
    public static void main(String[] args) {
        ClassPathResource res = new ClassPathResource("spring.cfg.xml");
        XmlBeanFactory factory = new XmlBeanFactory(res);
        DemoImple d1 = (DemoImple) factory.getBean("db");
        System.out.print(d1.sayHello());
    }
}
```

Constructor injection:

1. If spring containers uses parameterized constructor to create spring bean class object and to set values to bean properties as initial values then it is called as constructor injection.
2. Constructor injection supports injecting values to all types of bean properties.

3. If properties of bean class [few or all properties] are configured for constructor injection then spring bean container creates spring bean class object by using parameterised constructor.
4. If all properties of spring bean class are configured only for setter injection then spring container creates spring bean class object by using zero parameterized constructor.
5. Using <constructor-arg> tag in spring configuration file the container performs constructor injection on bean properties.
6. While configuring bean properteis for constructor injection we must configure them in spring configuration file in the same order or sequence as they are placed in the constructor parameters.
7. On the same bean property of bean class if we configure both setter and constructor injection then the value injected through setter injection gets activated because setter methods gets executed after execution of constructor. So that the values of constructor injection gets overridden with values of setter injection.
8. We can configure constructor argument values in 4 ways:
    1. Sequentially as specified in order of constructor
    2. By specifying datatype [here no two arguments should contain same data type]
    3. Using index of constructor arguments. [index of arguments starts from 0 [zero]].
    4. Combination of above 3.
9. Based on no. of <constructor-arg> tags specified in the bean class the same no. of parameterised constructor will be called by spring container.

    Eg:- If <constructor-arg> tag is placed for three times then three parameterised constructor gets executed.
10. Constructor injection is quick than setter injection. Because constructor gets executed at the time of object creation and setter methods gets executed after object creation.

Constructor vs. Setter injection: Use constructor injection when you have to inject one or few properties and prefer setter injection when you have to inject more proeprties.

Note:

1. Its not mandatory that all the bean class properties to be configured for dependency injection.
2. Maximum predefined classes supplied by spring are designed to support Setter injection.

//Example:Constructor Injection
//Demo.java

```
public class  Demo
{
        private String  message;
        public Demo() {} //dummy constructor
        public Demo(String message)  //one parameterised constructor
        {
                this.message = message;
        }
```

```
900         public String toString()
901         {
902                 return message;
903         }
904 }
```

spring.cfg.xml

```
906     <bean  id="demo"   class="Demo" >
907                 <constructor-arg>
908                         <value>Hello</value>
909                 </constructor-arg>
910     </bean>
```

Note:

1. Here all the tags like <value>, <ref>. <props>, <list>, <set>, <map> will remain same. But instead of using <property> tag we are making use of **<constructor-arg>** tag.
2. Client program is as usual.
3. As we are passing only one constructor argument in <bean> tag it invokes one parameterized constructor.

```
917 //Exmple: Invoking Multiple Parametersized constructor
918 public class Bean
919 {
920         private int a;
921         private float b;
922         private String c;

923         public Bean(){}

924         public Bean(int a, float b, String c)
925         {
926                 this.a = a;
927                 this.b = b;
928                 this.c = c;
929         }

930         public String toString()
931         {
932                 return a +" "+b+" "+c;
933         }
934 }
935 //spring.cfg.xml
936 //arguments in sequence

937         <bean id = "be" class = "Bean">
938                 <constructor-arg>
939                     <value>1</value>
940         </constructor-arg>
941         <constructor-arg>
942                 <value>1.1</value>
943         </constructor-arg>
```

```
        <constructor-arg>
            <value>Gmg</value>
        </constructor-arg>

</bean>
```

//Note:

1. In above style we have passed all the values in the form of sequence.
2. It means first <constructor-arg> value "1" will be given to first paramter of constructor.
3. second <constructor-arg> value "1.1" will be given to second paramter of constructor.
4. third <constructor-arg> value "Gmg" will be given to third paramter of constructor.

```
//spring.cfg.xml
//constructor arguments by using data type
    <bean id = "be" class = "Bean">
        <constructor-arg type = "float" >
            <value>1.1</value>
        </constructor-arg>

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

        <constructor-arg type = "int" >
            <value>1</value>
        </constructor-arg>
    </bean>
```

Note:

1. First argument of "float" data type will be given to float parameter of constructor
2. Scond argument of "java.lang.String" type will be given to String parameter of constructor
3. Third argument of "int" type will be given to int parameter of constructor.
4. But if the constructor has more than one parameter with same data type it will create error message due to ambiguity [confusion].

```
//spring.cfg.xml
//constructor arguments by using index
    <bean id = "be" class = "Bean">
        <constructor-arg index = "1">
            <value>1.1</value>
        </constructor-arg>

        <constructor-arg index = "2">
            <value>Gmg</value>
        </constructor-arg>

        <constructor-arg index = "0">
            <value>1</value>
        </constructor-arg>
    </bean>
```

Note:

1. Index always starts with 0 [zero]
2. 0^{th} index argument value will be given to 1^{st} parameter
3. 1^{st} index argument value will be given to 2^{nd} parameter
4. 2^{nd} index argument value will be given to 3^{rd} parameter

//spring.cfg.xml
//constructor arguments by using sequence, data type & index combination

```
<bean id = "be" class = "Bean">
        <constructor-arg index = "0" type ="int" >
                <value>1</value>
        </constructor-arg>

        <constructor-arg index = "1" type = "float">
                <value>1.1</value>
        </constructor-arg>

        <constructor-arg index = "2" type = "java.lang.String">
                <value>Gmg</value>
        </constructor-arg>
</bean>
```

Note:

1. We can make use of any of the above 4 styles.
2. But combination of sequence, index & type is the best way of configuring parameters. It will avoid confusion & increases readability of program. Due to this maintenance of application will become easy.
3. Client program is asusual.

Example: Constructor injection with different bean property types

//DemoInter.java

```
public interface DemoInter {
        public String sayHello();
}
```

//DemoImple.java

```
public class DemoImple implements DemoInter{
        private float salary;
        private TestBean tb = null;
        private Date dt = null;
        private List fruits = null;
        private List veg = null;
        private Set phone_nos = null;
        private Set emails = null;
        private Map perDetails = null;
        private Map capitals = null;
        private Properties faculties = null;
        private String courses[];
        private int runs[];
```

```
1028        public String sayHello() {

1029                return "Hello Friends... Gmg. :) "+
1030                " salary = "+salary+
1031                " tb = "+tb.toString()+
1032                " dt = "+dt.toString()+
1033                " fruits = "+fruits.toString()+
1034                " veg = "+veg.toString()+
1035                " phones = "+phone_nos.toString()+
1036                " emails = "+emails.toString()+
1037                " perDetails = "+perDetails.toString()+
1038                " capitals = "+capitals.toString()+
1039                " faculties = "+faculties.toString()+
1040                " courses = {"+courses[0]+" "+courses[1]+"}"+
1041                " runs = {" + runs[0]+" "+runs[1]+"}";
1042        }

1043        public DemoImple(float salary, TestBean tb, Date dt, List fruits, List veg,
1044                        Set phone_nos, Set emails, Map perDetails, Map capitals,
1045                        Properties faculties, String[] courses, int[] runs)
1046        {
1047                this.salary = salary;
1048                this.tb = tb;
1049                this.dt = dt;
1050                this.fruits = fruits;
1051                this.veg = veg;
1052                this.phone_nos = phone_nos;
1053                this.emails = emails;
1054                this.perDetails = perDetails;
1055                this.capitals = capitals;
1056                this.faculties = faculties;
1057                this.courses = courses;
1058                this.runs = runs;
1059        }
1060        public DemoImple() {} //dummy constructor
1061 }
```

//Note:

1. Same as the example of different property types with setter injection. There we made use of <property> tag. Now we are making use of <constructor-arg> tag. Interanally body of the <property> tag or <constructor-arg> tag will remain same. The only change is with outer tag.
2. Here we are following sequence while passing arguments to constructor.

//spring.cfg.xml

```
1069 <?xml version="1.0" encoding="UTF-8"?>
1070 <beans
1071        xmlns="http://www.springframework.org/schema/beans"
1072        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
1073        xsi:schemaLocation="http://www.springframework.org/schema/beans
1074 http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">
```

```
<bean id ="dat" class = "java.util.Date"/>

<bean id ="tbean" class = "TestBean">
        <property name="msg">
                <value>Hello</value>
        </property>
</bean>
<bean id = "db" class = "DemoImple">
        <constructor-arg>
                <value>99.9</value>
        </constructor-arg>
        <constructor-arg>
                <ref bean = "tbean"/>
        </constructor-arg>
        <constructor-arg>
                        <ref bean = "dat" />
        </constructor-arg>
        <constructor-arg>
                <list>
                        <value>Apple</value>
                        <value>Banana</value>
                        <value>Orange</value>
                </list>
        </constructor-arg>
        <constructor-arg>
            <list>
                <value>Carrot</value>
                <value>Potato</value>
                <value>Mushrooms</value>
                <!--  any objects can be stored in a List-->
                <ref bean = "dat"/>
                <ref bean = "tbean"/>

                <value>Tomato</value>
             </list>
        </constructor-arg>
        <constructor-arg>
                <set>
                        <value>7569670669</value>
                        <value>9502166767</value>
                </set>
        </constructor-arg>
        <constructor-arg>
                <set>
                        <value>kanakavaraprasad@gmail.com</value>
                        <value>s.varaprasad@yahoo.co.in</value>
                </set>
        </constructor-arg>
```

```
1121            <constructor-arg>
1122                <map>
1123                    <entry>
1124                        <key>
1125                            <value>Sai</value>
1126                        </key>

1127                        <value>Universal King</value>
1128                    </entry>

1129                    <entry>
1130                        <key>
1131                            <value>Kanakadhar</value>
1132                        </key>
1133                        <value>Service Provider</value>
1134                    </entry>
1135                </map>
1136            </constructor-arg>
1137            <constructor-arg>
1138                <map>
1139                    <entry>
1140                        <key>
1141                            <value>Sathya</value>
1142                        </key>

1143                        <value>Boon for learners</value>
1144                    </entry>

1145                    <entry>
1146                        <key>
1147                            <ref bean = "dat"/>
1148                        </key>

1149                        <ref bean = "tbean"/>
1150                    </entry>
1151                </map>
1152        </constructor-arg>
1153            <constructor-arg>
1154                <props>
1155                    <prop key="Kanakadhar">Java Faculty</prop>
1156                    <prop key="Mahesh">.net Faculty</prop>
1157                </props>
1158            </constructor-arg>
1159            <constructor-arg>
1160                <list>
1161                    <value>Java</value>
1162                    <value>.net</value>
1163                    <value>Oracle</value>
1164                </list>
1165            </constructor-arg>
```

```
            <constructor-arg>
                <list>
                    <value>30</value>
                    <value>40</value>
                </list>
            </constructor-arg>
        </bean>
</beans>
```

Note: Client program is asusual.

Factory method: A method of a java class which is capable of constructing and returns its own class object.

When spring bean class has private constructor we cannot create its object from outside. Then we can give instructions to spring container to create the object of bean class by using factory method.

Note:

1. Factory methods can be of static or not static [instance method]
2. Factory method internally calls constructor at the time of object creation. No problem will be there even with private constructors. Because private constructors can be accessed from inside the class.
3. Factory methods should be public methods. So that can be accessed from outside the class.

Eg of static factory methods:

1. Class c = Class.forName("ClassName");
2. Thread t = Thread.currentThread();
3. Calendar cl = Calendar.getInstance();

Eg. on not static factory method [insanmce method]

```
String s = new String("OK");
String s1 = s.concat("Hello");  //instance factory method.
```

Instructions to spring container to call factory method:

1. We should use **factory-method** attribute of <bean> tag to configure the factory method to be called to create the object.
2. If factory method expects any arguments then pass required arguments using <constructor-arg> tag.

Factory Class: A class method which can return you another class object is called as Factory Class.

Eg:

```
Intege i1 = new Integer(100);
String s1= i1.toString();
```

Note: Here by using toString() of java.lang.Integer class we are getting java.lang.String. It is returning this object by creating immediately. So we can call Integer class as factory class.

Approaches to create spring bean class objects:

1. zero or parameterized constructor
2. static factory method of bean class
3. instance factory method of factory bean class

Example: Static Factory Method

//DemoInter.java

```
public interface DemoInter
{
        public String sayHello();
}
```

// DemoBean.java

```
import java.util.Calendar;
public class DemoBean implements DemoInter
{
        private Calendar cl = null;
        private TestBean tb = null;

        public DemoBean() { }

        public void setCl(Calendar cl) {
                this.cl = cl;
        }

        public void setTb(TestBean tb) {
                this.tb = tb;
        }

        public String sayHello() {

                return "tb = "+tb.toString()+"   cl = "+cl.toString();
        }
}
```

//TestBean.java

```
public class TestBean {

        String msg;

        public TestBean() {}

        private TestBean(String msg) {
                this.msg = msg;
                System.out.println("Private Constructor invoked");
        }
```

```
1238        //static factory method
1239        public static TestBean getTestBean(String msg)
1240        {
1241            System.out.println("static factory method invoked");

1242            return new TestBean(msg);
1243        }

1244        public String toString() //to dispaly results
1245        {
1246            return msg;
1247        }
1248    }
1249    //spring.cfg.xml
1250    <?xml version="1.0" encoding="UTF-8"?>
1251    <beans
1252        xmlns="http://www.springframework.org/schema/beans"
1253        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
1254        xsi:schemaLocation="http://www.springframework.org/schema/beans
1255    http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">

1256        <bean id = "cal" class = "java.util.Calendar" factory-method = "getInstance"/>

1257        <bean id = "t1" class = "TestBean" factory-method = "getTestBean">
1258            <constructor-arg>  <!-- passing argument to factory method -->
1259                <value>Gmg. my dear friends...</value>
1260            </constructor-arg>
1261        </bean>

1262        <bean id = "db" class = "DemoBean">
1263            <property name = "cl">
1264                <ref bean = "cal"/>
1265            </property>

1266            <property name = "tb">
1267                <ref bean = "t1"/>
1268            </property>
1269        </bean>

1270    </beans>
1271    //DemoClient.java
1272    public class DemoClient {

1273        public static void main(String[] args) {
1274            ClassPathResource res = new ClassPathResource("spring.cfg.xml");
1275            XmlBeanFactory factory = new XmlBeanFactory(res);
1276            DemoBean d1 = (DemoBean) factory.getBean("db");
1277            System.out.print(d1.sayHello());
1278        }
1279    }
```

Note:

1. The spring container uses factory method getInstance() to create bean class object for java.util.Calendar.
2. Spring container uses "getTestBean(String)" as factory method to create test bean class object.
3. Spring container uses zero parameterised constructor to create DemoBean class object.

//Example: Instance Factory Method

//DemoInter .java

```
public interface DemoInter {
        public String sayHello();
}
```

// DemoBean.java

```
public class DemoBean implements DemoInter{

        private TestBean tb = null;

        public DemoBean() {
                System.out.println("zero parametrized constructor of DemoBean class");
        }

        public void setTb(TestBean tb) {
                this.tb = tb;
        }

        public String sayHello() {

                return "tb = "+tb.toString();
        }

        //instance factory method
        public TestBean getTestBean(String msg)
        {
                System.out.println("demo bean factory method of DemoBean class");
                return new TestBean(msg);
        }
}
```

//TestBean.java

```
public class TestBean {
        String msg;

        public TestBean() {} //dummy constructor

        TestBean(String msg) {
                this.msg = msg;
                System.out.println("Constructor of TestBean class invoked");
        }
```

```
        public String toString() //to dispaly results
        {
                return msg;
        }
}
```

//spring.cfg.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans
        xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">

        <bean id = "t1" factory-bean = "db" factory-method = "getTestBean">
                <constructor-arg>  <!-- passing argument to factory method -->
                        <value>Gmg. my dear friends...</value>
                </constructor-arg>
        </bean>

        <bean id = "db" class = "DemoBean">
                <property name = "tb">
                        <ref bean = "t1"/>
                </property>
        </bean>
</beans>
```

//DemoClient .java

```
public class DemoClient {
        public static void main(String[] args) {
                ClassPathResource res = new ClassPathResource("spring.cfg.xml");
                XmlBeanFactory factory = new XmlBeanFactory(res);
                DemoBean d1 = (DemoBean) factory.getBean("db");
                System.out.print(d1.sayHello());
        }
}
```

Note:

1. Making spring container to create spring bean class object through non static methods [instance methods].
2. While configuring spring bean class if class attribute is not there and factory-bean, factory-method attributes are there then spring container uses instance factory method of specified factory bean class to create spring bean class object.

Application Context container is enhancement of bean factory container with some advantages. Its features are as follows:

1. Pre initialization of beans by default. [Early loading]. It means in our spring configuration file we may have any no. of bean tags. Objects will be created sequentially for each and every bean class as specified in the sequence of <bean> tags in xml file.
2. Ability to read values of bean properties from properties file.

3. supports Internationalization (I18n)
4. Gives the ability to work with events & listeners.
5. org.springframework.context.ApplicationContext interface is the sub interface of org.springframework.beans.factory.BeanFactory interface.
6. Activating application context interface is nothing but creating object of a class that implements org.springframework.context.ApplicationContext interface.
7. There are three regularly used implementation classes of application context interface. By creating object for any of these three classes we can activate ApplicationContext container.
8. This container is available in JEE module.
9. **org.springframework.context.support.FileSystemXmlApplicationContext:** It activates application context container by locating given spring configuration file in the specified path.

   Ex: FileSystemXmlApplicationContext ctx = new FileSystemXmlApplicationContext("c:\f1\spriong.cfg.xml")
10. **org.springframework.context.support.ClassPathXmlApplicationContext:** It activates application context container by locating spring configuration file in the same working directory or from jar files added in the classpath.

    Ex: ClassPathXmlApplicationContext ctx = new ClassPathXmlApplicationContext("spring.cfg.xml");
11. **org.springframework.context.support.XmlWebApplicationContext:** this class activates application context container by locating spring configuration file in deployment directory structure of web application by default in WEB-INF folder.

    Ex: XmlWebApplicationContext ctx = new XmlWebApplicationContext("spring.cfg.xml");
12. In real time applications we can find regularly working with Application Context container rather than working with Bean Factory container.
13. Application context container can perform all modes of dependency injection like bean factory container.
14. Application context container performs pre instantiation on all singleton scoped on spring bean classes of spring configuration file at the moment of application context container gets activated.
15. Pre-instantiation means creating spring bean class objects immediately after the application context container activation.
16. BeanFactory cannot perform this pre instantiation on spring bean classes.
17. when we call factory.getBean(_) then bean factory container immediately creates the object and uses the same for further requests on same id-value.
18. When we call ctx.getBean(_) then application context container gives access to bean id related spring bean class object which is created at pre-instantiation process.
19. Application context container can perform pre-instantiation only on singleton scoped spring bean classes that are configured in spring configuration file.
20. If we give scope value as prototype then object will not be created for bean at the time of pre-instantiation.

21. If the object is not created at pre-instantiation then it will be created after receiving the call along with its dependent objects.
22. If singleton scope bean class property has prototype scoped bean class object as dependent value then the application context container also creates prototype scoped spring bean class object during pre instantiation process along with singleton scope bean class object.
23. It happens to satisfy dependency injection needs done on singleton scope bean class properties.

Example: Creation of our own singleton java class.

```
//Designing our own singleton java class

//Test.java
public class Test
{
	private static Test t = null;
	//private constructor-so no one can create object from outside the class
	private Test()
	{
		System.out.println("Private constructor of Test class invoked");
	}
	//static factory method
	public static Test getTest()
	{
		//singleton logic
		if(t == null)
			t = new Test();
		return t;
	}
}
```

Note: Also override clone() method of super class Object and throw CloneNotSupportedException.

```
//Client.java
class Client
{
	public static void main(String[] args)
	{
		Test t1 = Test.getTest();
```

```
        Test t2 = Test.getTest();
        Test t3 = Test.getTest();

        System.out.println(t1);
        System.out.println(t2);
        System.out.println(t3);
    }
}
```

Note: We have requested the same class object 3 times. But only one object gets created & the same object will be given for all 3 times.

Scopes of Bean class objects under ApplicationContext container:

We can make spring container to keep the created spring bean objects in different scopes. They are:

1. singleton [**default scope**]
2. prototype
3. request
4. session
5. global session

singleton: container makes spring bean class object as **shared object for multiple calls** given to factory.getBean() with bean id.

Eg:

<bean id = “bd” class = “BeanDemo” **scope = “singleton”**/>

The client application may call factory.getBean(“bd”) for multiple times. Still the spring container creates and returns only one DemoBean class object.

Singleton java class means the java class that allows to create only one object per jvm. It means the class has restriction related logic.

Note: In the above example **BeanDemo** class need not to be designed as singleton java class. But container creates only one object for DemoBean class. Here it’s the responsibility of spring container.

When spring bean scope is singleton the spring container never makes spring bean class as singleton java class but spring container restricts itself and creates only one object for spring bean class though factory.getBean(id) is called for multiple times.

Prototype: It gives an independent spring bean class object for each call to factory.getBean(id); **Eg:**
<bean id = “db” class = “DemoBean” scope = “prototype”/>

“request”, “session”, “global session” are used for web applications.

request: Only one object will be used from beginning to end of the same request.

session: from beginning to expiry of same session single object will be used.

global session: Its commonly shared by the entire web application.

Note: When we give setter injection we should not give only parameterized constructor in the class.

Because when no <constructor-arg> tags are available in spring configuration file then spring container tries to create object by using default [zero parameterized] constructor. If its not available then immediately error arises. When no constructors (zero or parameterized) are available in the class then automatically compiler creates a default and zero parameterized constructor.

//Example: First application with ApplicationContext or JEE or Advanced Container

//Client steps:

1. Locate Spring-Configuration file + Activate ApplicationContext Container + Read Spring-Configuration file
2. Request object of business-logic class.
3. Call business-logic methods.
4. Close container

//DemoInter.java

```
public interface  DemoInter
{
        public String wish(String uname);
}
```

//**DemoInterImpl.java**

```
public class  DemoInterImpl  implements DemoInter
{
        private String  message;
        public DemoInterImpl()
        {
                System.out.println("Constructor executed");
        }
         public void setMessag(String s)
        {
                message=s;
        }
        public String wish(String uname)
        {
                return   message+" "+uname+" Have a great day";
        }
}
```

//spring.cfg.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans
        xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">

  <bean  id="demo"   class="DemoInterImpl" >
                <property  name="messag">
                        <value>Hello</value>
                </property>
     </bean>
</beans>
```

//DemoClient.java

```
import org.springframework.context.support.ClassPathXmlApplicationContext;
public class   DemoClient
{
      public static void main(String args[])
      {
         ClassPathXmlApplicationContext  ctx = new
         ClassPathXmlApplicationContext("spring.cfg.xml");
         DemoInter d1=(DemoInter)ctx.getBean("demo");
         DemoInterImpl d2=(DemoInterImpl)ctx.getBean("demo");
         System.out.println(d1.wish("Sai"));
         ctx.close();
      }
}
```

Note: We are requesting the same class object twice. But container creates only one object & gives the same object for every request.

Example: Prototype scope

Note:

1. Same as above example. We are giving a new attribute for our <bean> tag as shown below in spring configuration file.
2. If we donot specify any scope by default the scope will be **singleton.**
3. If the scope is prototype:

1. for every request a separate object will be created.
2. Without recieving the request object will not be created by container.

```
<bean id="demo" class="DemoInterImpl" scope="prototype">
        <property name="messag">
                <value>Hello friends... Gmg. :)</value>
        </property>
</bean>
```

Interface Injection:

1. If sring container performs dependency injection on bean properties by implementing the XxxAware interfaces then it is called as interface injection.
2. All these XxxAware interfaces are predefined interfaces supplied by spring api on one bean spring bean class property.
3. We can perform all 3 types of dependency injection.
4. We can't use interface injection to inject our own values.
5. Interface injection can perform special values injection given by spring container.

Predfined interfaces used for dependency injection:

1. org.springframework.beans.factory.BeanNameAware interface makes spring bean class getting its bean id given in spring configuration file through interface injection.
2. org.springframework.beans.factory.BeanFactoryAware interface injects the current underlying bean factory container to spring bean class through interface injection.
3. org.springframework.context.ApplicationContextAware injects the current underlying application context container to spring bean class.

Note:

1. While performing interface injection on spring bean class the spring bean class can't act as "pojo class".
2. Interface injection is useful to make spring bean to know about itself and other spring beans managed by the container.

Example: Interface Injection

//Demo.java

```
public interface Demo {
        String hi();
}
```

//DemoBean.java

```
import org.springframework.beans.BeansException;
import org.springframework.beans.factory.BeanNameAware;
import org.springframework.context.ApplicationContext;
import org.springframework.context.ApplicationContextAware;
```

```
1579 public class DemoBean implements Demo, BeanNameAware, ApplicationContextAware{
1580         String msg, bname;
1581         ApplicationContext ctx = null;
1582         public void setMsg(String msg) {
1583                 this.msg = msg;
1584         }
1585         public void setApplicationContext(ApplicationContext ctx)
1586                         throws BeansException {
1587                 this.ctx = ctx;
1588         }
1589         public void setBeanName(String bname) {
1590                 this.bname = bname;
1591         }
1592         public String hi() {
1593                 System.out.println("Logical current bean name is: "+bname);
1594                 System.out.println("No. of beans managed by underlying container is:
1595 "+ctx.getBeanDefinitionCount());
1596                 System.out.println("Current bean is singleton: "+ctx.isSingleton(bname));
1597                 System.out.println("Current bean is proptotype bean: "+ctx.isPrototype(bname));
1598                 return msg;
1599         }
1600 }
1601 // applicationContext.xml
1602 <?xml version="1.0" encoding="UTF-8"?>
1603 <beans xmlns="http://www.springframework.org/schema/beans"
1604         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
1605         xsi:schemaLocation="http://www.springframework.org/schema/beans
1606 http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">
1607         <bean id = "db" class = "DemoBean">
1608                 <property name="msg">
1609                         <value>Hi Friends... Gmg. Njoy Spring concepts n coding. :)</value>
1610                 </property>
1611         </bean>
1612         <bean id = "dt" class = "java.util.Date" scope = "prototype"/>
1613         <bean id = "dt1" class = "java.util.Date"/>
1614 </beans>
```

//DemoClient.java

```
import org.springframework.context.support.ClassPathXmlApplicationContext;
import java.util.Date;
public class   DemoClient
{
      public static void main(String args[])
      {
         ClassPathXmlApplicationContext  ctx = new
         ClassPathXmlApplicationContext("applicationContext.xml");


         Demo d1=(Demo)ctx.getBean("db");
         Date d2=(Date)ctx.getBean("dt");
         //DemoImpl d2=(DemoImpl)ctx.getBean("db");
                  System.out.println(d1.hi());
                  System.out.println(d2);
      }
}
```

Reading Properties File: Place holder represents bean property value that should be collected from properties file based on the given key name dynamically at runtime of the program.

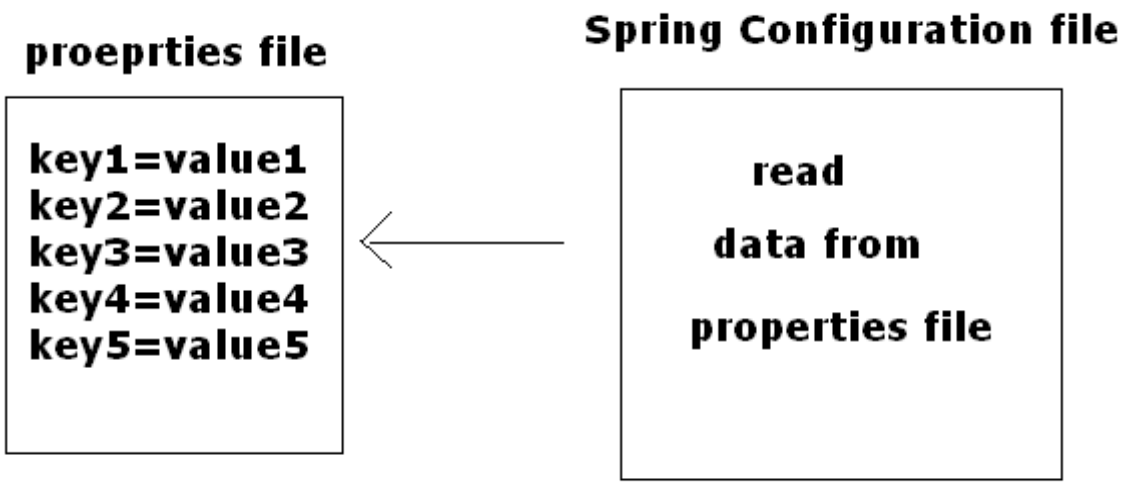
Purpose of properties file in spring application: properties files and xml files are given to make java application code as flexible by passing input values of application from outside the application.

Note:

1. Work with properties files only when our spring application is forced to work with existing properties file. Don't create new properties files because the same flexibility can be achieved with xml file (spring configuration file) also.
2. The standard principle of s/w industry is "don't hard code any values that can be modified in future or changed for regular intervals".

   Eg: jdbc driver class name, url, username, password etc..,

Example: Reading a values from properties file & writing to spring configuration file.



Note: In properties file we can specify comments by using # or !

//data.properties

```
#demo.properties
our.name = Dharani
our.age = 25
our.address = Hyd
our.email = dharani@gmail.com
```

//DemoBean.java

```
public class DemoBean {
	String name, email, address;

	int age;
	public void setName(String name) {
		this.name = name;
	}
	public void setEmail(String email) {
		this.email = email;
	}
	public void setAddress(String address) {
		this.address = address;
	}
	public void setAge(int age) {
		this.age = age;
	}
```

```
1666        public String hi()
1667        {
1668                return name+" "+age+" "+email+" "+address;
1669        }
1670  }
1671  //applicationContext.xml
1672  <?xml version="1.0" encoding="UTF-8"?>
1673  <beans
1674        xmlns="http://www.springframework.org/schema/beans"
1675        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
1676        xsi:schemaLocation="http://www.springframework.org/schema/beans
1677  http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">

1678        <bean id = "propConfig" class =
1679  "org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
1680              <property name="location" value = "pf\demo.properties">
1681              <!--    <value>demo.properties</value>  -->
1682              </property>
1683        </bean>

1684        <bean id = "db" class = "DemoBean">
1685              <property name="name">
1686                    <value>${our.name}</value>
1687              </property>

1688              <property name="email">
1689                    <value>${our.email}</value>
1690              </property>
1691              <property name="address">
1692                    <value>${our.address}</value>
1693              </property>
1694              <property name="age">
1695                    <value>${our.age}</value>
1696              </property>
1697        </bean>
1698  </beans>
```

//DemoClient.java

```
import org.springframework.context.support.ClassPathXmlApplicationContext;
public class   DemoClient
{
      public static void main(String args[])
      {
         ClassPathXmlApplicationContext  ctx = new
         ClassPathXmlApplicationContext("applicationContext.xml");
         DemoBean d = (DemoBean) ctx.getBean("db");
         System.out.println(d.hi());
      }
}
```

Example: Reading data from multiple properties files

#pf1.properties

```
name = Dharani
age = 25
```

#pf2.properties

```
address = Hyd
email = dharani@gmail.com
name = Sathya
```

Note:

1. "name" property is there in both properties files. So first property file value will be replaces with second property file value.
2. DemoBean.java & DemoClient.java files are same as above example.

//applicationContext.xml

```
      <bean id = "propConfig" class =
      "org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
            <property name="locations">
                  <list>
                        <value>pf1properties</value>
                        <value>pf2.properties</value>
                  </list>
            </property>
      </bean>
```

```
<bean id = "db" class = "DemoBean">
        <property name="name">
                <value>${name}</value>
        </property>
        <property name="email">
                <value>${email}</value>
        </property>
        <property name="address">
                <value>${address}</value>
        </property>
        <property name="age">
                <value>${age}</value>
        </property>
</bean>
```

Example: Reading multiple configuration files

//applicationContext1.xml

```
<beans>
        <bean id = "t1" class = "TestBean">
                <property name="msg">
                        <value>Hello friends Gmg.</value>
                </property>
        </bean>
</beans>
```

//applicationContext2.xml

```
<beans>
        <bean id = "d1" class = "DemoBean">
                <property name="tb">
                        <ref bean = "t1"/>
                </property>
        </bean>
</beans>
```

Note: "applicationContext2.xml" is reading identification name of "applicationContext1.xml" file. It means we may have any number of configuration files for the purpose of easy organization, still programaticdally we can consider them as a single file.

```
1767  //DemoBean.java
1768  public class DemoBean {
1769          TestBean tb = null;
1770          public void setTb(TestBean tb) {
1771                  this.tb = tb;
1772          }
1773          public String toString()
1774          {
1775                  return tb.msg;
1776          }
1777  }
1778  // TestBean.java
1779  public class TestBean {
1780          String msg;
1781          public void setMsg(String msg) {
1782                  this.msg = msg;
1783          }
1784          public String toString()
1785          {
1786                  return msg;
1787          }
1788  }
1789  //DemoClient.java
1790  import org.springframework.context.support.ClassPathXmlApplicationContext;

1791  public class   DemoClient
1792  {
1793        public static void main(String args[])
1794        {
1795           String cfg[] = {"applicationContext2.xml", "applicationContext1.xml"};
1796           ClassPathXmlApplicationContext  ctx = new ClassPathXmlApplicationContext(cfg);
1797           DemoBean d = (DemoBean) ctx.getBean("d1");
1798           System.out.println(d);
1799        }
1800  }
```

Internationalization (I18n):

1. Making our application projects working for multiple locales with flexibility in presentation logic to pass display labels of input fields or any information from outside the application is called applying **I18n** on the project.
2. The project i.e., enabled with I18n can give service to multiple client organizations which may belong to multiple communities without changing source code.

   Locale = language + country

   Examples for Language code & country code:

   - En-us [English – United States]
   - En-au [English – Australia]
   - En-ca [English – Canada]
   - En-ie [English Ireland]
   - En-nz [English new zealand]
   - En-zw [English – Zimbabwe]
   - En-gb [English – United Kingdom]
   - Fr-be [French – Belgium]
   - Fr-ca [French – Canada]
   - Fr [French france]
   - Fr-ch [French Switzerland]
3. All the above locales codes are given by "ISO" we can collect them from "IE" [Internet explorer] browser window settings as follows: Tools à Internet options à languages à Add [it shows you multiple countries with multiple languages. So we can choose what we need for our applications]
4. Here we write a separate properties file for each language and we furnish all the labels in it for display of labels based on internet options of browser.
5. In java application we use Locale class & ResourceBundle class of java.util package to apply the effect of I18n.
6. While preparing multiple locale specific properties files the base file must be there because when no matching file is available then values will be taken from this **base file**.
7. Org.springframework.conext.support.**ResouceBundleMessageSource** class makes the underlying application context container recognizing multiple locale properties files on I18n that are given to the application.
8. In order to make the container to get message from properties file we need to call getMessage() method on the container object.
9. For **getMessage()** method we have to pass 4 argumetns. They are:
   1. key in properties file
   2. java.lang.Object class array to pass argument values to place holders available in property values. If place holders are not there then we can pass "null".as a value.

3. **Default message:** when message from given key is not collectable default message will be displayed.
4. **java.util.Locale** class object with language and country code.

Example: Spring Internationalization

#app.preoperties [base file]

str1 = save - {0}english {1}

str2 = clear - english

str3 = delete - english

str4 = cancel - english

Note:

1. "Str1" property has two place holders.
2. Every poperty value can have place holders
3. Always place holders starts with zero.
4. For every property value place holder values can be supplied through string array variable.

#app_en_ca.preoperties [english canada]

str1 = save - {0} english {1} canada

str2 = clear - english canada

str3 = delete - english canada

str4 = cancel - english canada

#app_en_gh.preoperties [uk]

str1 = save -{0} english {1} uk

str2 = clear - english uk

str3 = delete - english uk

str4 = cancel - english uk

#app.preoperties [french belgium]

str1 = save - {0} french {1} belgium

str2 = clear - french belgium

str3 = delete - french belgium

str4 = cancel - french belgium

//applicationContext.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans
        xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">
```

```
        <bean id = "messageSource" class =
        "org.springframework.context.support.ResourceBundleMessageSource">
                <property name="basename">
                        <value>app</value>
                </property>
        </bean>
</beans>
```

//I18nApp.java

```
import java.util.Locale;
import org.springframework.context.support.ClassPathXmlApplicationContext;
public class I18nApp {
        public static void main(String args[])throws Exception
        {
                Locale l = new Locale(args[0], args[1]);
                ClassPathXmlApplicationContext  ctx = new
        ClassPathXmlApplicationContext("applicationContext.xml");
                String msg = ctx.getMessage("str1", new String[]{"hi", "Sai"}, "default msg", l);
                msg +=" "+ ctx.getMessage("str2", null, "default msg", l);
                msg +=" "+ ctx.getMessage("str3", null, "default msg", l);
                msg +=" "+ ctx.getMessage("str4", null, "default msg", l);
                System.out.println("Message is: "+msg);
        }
}
```

Event Handling: Event is specific action performed on the component or object. To handle events and execute same logic every time we can depend upon listeners. These listeners provide event handling methods to handle the events.

All events are objects and all listeners are interfaces in java environment

In order to perform event handling four details are required:

1. Source object [Eg: Button component]
2. Event [Eg: ActionEvent]
3. Listener [Eg: ActionListener]
4. Event handling methods [Eg: public void actionPerform()]

In spring environment event handling is given to notify when the spring container is activated or deactivated in spring based applications or other java applications where spring is also used.

This event handling helps the programmer to notice how much time the spring related business logic is executed in the project.

To perform event handling on spring application we need the following details:

1.source object [application context container]

2.event [application event]

3. listener [application listener]

4.event handling methods on application event

ApplicationContext container supports 3 events:

1. ContextRefreshedEvent
2. RequestHandledEvent[Supported only in web applications. It gets activated for every request]
3. ContextClosedEvent

Note:

1. Event handling on spring container can be performed without touching existing java source code of application.
2. For more information on this refer 3.8.3 Events topic of spring PDF file.

Example: Event Handling

```
public interface  DemoInter
{
        public String wish(String uname);
}
public class  DemoInterImpl  implements DemoInter
{
        private String  message;
        public DemoInterImpl()
        {
                System.out.println("Constructor executed");
        }
         public void setMessag(String s)
        {
                message=s;
        }
        public String wish(String uname)
        {
                return   message+" "+uname+" Have a great day";
        }
}
```

```
// Helper.java
import java.util.Date;
import org.springframework.context.ApplicationEvent;
import org.springframework.context.ApplicationListener;
public class Helper implements ApplicationListener {
        long startTime, endTime;
        public void onApplicationEvent(ApplicationEvent ae)
        {
                System.out.println("control in onApplicationEvent method");
                System.out.println(ae.toString());
                int index = ae.toString().indexOf("ContextRefreshedEvent");
                System.out.println("Index is: "+index);
                if(index != -1)
                {
                        startTime = System.currentTimeMillis();
                        System.out.println("Container started at "+new Date().toString());
                }
                else if(ae.toString().indexOf("ContextClosedEvent") != -1)
                {
                        endTime = System.currentTimeMillis();
                        System.out.println("Container is Closed at "+new Date().toString());
                        System.out.println("Container is in activated mode for "+(endTime -
startTime)+" milli seconds");
                }
        }
}
//spring.cfg.xml
<?xml version="1.0" encoding="UTF-8"?>
<beans
        xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">
  <bean id = "help" class = "Helper"/>
  <bean  id="demo"   class="DemoInterImpl" >
                <property  name="messag">
```

```
                    <value>Hello</value>
                </property>
    </bean>
</beans>
```

//DemoClient.java

```
import org.springframework.context.support.ClassPathXmlApplicationContext;
public class   DemoClient
{
      public static void main(String args[])throws Exception
      {
         ClassPathXmlApplicationContext  ctx = new
         ClassPathXmlApplicationContext("spring.cfg.xml");

         DemoInter d1=(DemoInter)ctx.getBean("demo");
         //DemoInterImpl d2=(DemoInterImpl)ctx.getBean("demo");
         System.out.println(d1.wish("Sai"));
         Thread.sleep(2000);
         ctx.close(); //container gets closed now
      }
}
```

User defined life cycle methods: Spring bean class allows configuring our user defined methods to act like life cycle methods.

1. **init () life cycle method:** container calls this method automatically after dependency injection process is completed on bean properties.
2. **destroy() life cycle method:** Container calls this method just before destruction of spring bean class object created by spring container.

We can configure these life cycle methods in two ways:

1. Programmatic statements [by implementing the following interfaces]
    1. InitializingBean
        1. afterPropertiesSet()
    2. DisposableBean
        1. destroy()
2. Declarative statements
    1. <bean> attributes
        1. init-method
        2. destroy-method

As these life cycle methods are user defined methods their names should be configured during spring bean class configuration in spring configuration file by using **init-method** and **destroy-method** attributes of **<bean>** tag.

Note: We can give any names for these life cycle methods. But init & destroy are standard names. Any way what ever the name it is, it's mandatory to configure them in configuration file.

Init() method is useful to check whether appropriate values were injected to spring bean properties are not through dependency injection.

destroy() life cycle method is useful to nullify bean properties and to release non java resources associated with bean properties when spring bean class object is about to destroy.

Example: Programmatic initialization method

```
import org.springframework.beans.factory.InitializingBean;
public class Init implements InitializingBean{
        int p1;
        public Init()
        {
                System.out.println("In  constructor");
        }
        public void setP1(int p1)
        {
                System.out.println("Insetter method");
                this.p1 = p1;
        }
        public void afterPropertiesSet() throws Exception
        {
                System.out.println("in after proeprties set "+p1);
        }
}
```

//applicationContext.xml

```
<beans>
  <bean  id="in" class="Init">  <!-- init-method = "init" -->
        <property name = "p1" value = "10"/>
  </bean>
</beans>
```

//DemoClient.java

```
import org.springframework.context.support.ClassPathXmlApplicationContext;
public class DemoClient {
        public static void main(String[] args) {
```

```
                ClassPathXmlApplicationContext  ctx = new
        ClassPathXmlApplicationContext("applicationContext.xml");
                Init i1 = (Init) ctx.getBean("in");
                System.out.print(i1);
        }
}
```

Example: Programmatic destroy method

```
import org.springframework.beans.factory.DisposableBean;
public class Destroy implements DisposableBean{
        int p1;
        public Destroy() {
                System.out.println("In  constructor  of Destroy class");
        }
        public void setP1(int p1) {
                System.out.println("Insetter method");
                this.p1 = p1;
        }
        public void destroy() throws Exception {
                System.out.println("In  destructor of Destroy class");
        }
}
```

//applicationContext.xml

```
<beans>
        <bean  id="de" class="Destroy">
                <property name = "p1" value = "20"/>
        </bean>
</beans>
```

//DemoClient.java

```
import org.springframework.context.support.ClassPathXmlApplicationContext;
public class DemoClient {
        public static void main(String[] args) {
                ClassPathXmlApplicationContext  ctx = new
ClassPathXmlApplicationContext("applicationContext.xml");
                Destroy d = (Destroy) ctx.getBean("de");
                System.out.print(d);
                ctx.close();
```

```
        }
}
```

//Example: Declarative init() method

Note:

1. By using declarative statements we can configure any method as a init() method.
2. Need not to implement any interface in case of declarative approach.

//Init.java

```
public class Init{
        int p1;
        public Init() {
                super();
                System.out.println("In  constructor");
        }
        public void setP1(int p1) {
                System.out.println("In setter method");
                this.p1 = p1;
        }
        public void init()
        {
                System.out.println("in user defined call back method");
        }
}
```

//applicationContext.xml

```
<beans>
  <bean  id="in" class="Init" init-method = "init">
        <property name = "p1" value = "100"/>
  </bean>
</beans>
```

// DemoClient.java

```
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;
public class DemoClient {
        public static void main(String[] args) {
                ClassPathResource res = new ClassPathResource("applicationContext.xml");
```

```
                XmlBeanFactory factory = new XmlBeanFactory(res);
                Init i1 = (Init) factory.getBean("in");
                System.out.print(i1);
        }
}
```

//Example: Declarative destroy() method

Note:

1. By using declarative statements we can configure any method as a destroy() method.
2. Need not to implement any interface in case of declarative approach.

//Destroy.java

```
public class Destroy
{       int p1;
        public Destroy() {
                System.out.println("In  constructor  of Destroy class");
        }
        public void setP1(int p1) {
                System.out.println("In setter method");
                this.p1 = p1;
        }
        public void destroy() {
                System.out.println("In  destructor of Destroy class");
        }
}
```

//applicationContext.xml

```
<beans>
<bean  id="de" class="Destroy" destroy-method = "destroy">
        <property name="p1">
                <value>11</value>
        </property>
</bean>
</beans>
```

Note: Client program is same as above

Inner Beans: If we define a <bean> tag inside another bean tag then we call it as inner bean. Inner beans are used to provide security. If a class is declared as independent bean then the same class can be used by any other class. The same class if we declare as a inner bean then it can be used only by outer bean class.

Example:

```
public class First {
	int a;
	int b;
	public void setA(int a) {
		this.a = a;
	}
	public void setB(int b) {
		this.b = b;
	}
	@Override
	public String toString() {
		return a+" "+b;
	}
}
public class Second {
	First f;
	public void setF(First f) {
		this.f = f;
	}
	@Override
	public String toString() {
		return f.toString();
	}
}
```

//applicationContext.xml

```
<beans>
	 <bean id = "two" class = "Second">
		<property name="f">
			<bean class = "First">
				<property name="a">
					<value>10</value>
				</property>
				<property name="b">
					<value>20</value>
```

```
                                    </property>
                                </bean>
                        </property>
                </bean>
</beans>
```

//Client.java

```
import org.springframework.context.support.ClassPathXmlApplicationContext;
public class Client {
        public static void main(String[] args) {
                ClassPathXmlApplicationContext  ctx = new
        ClassPathXmlApplicationContext("applicationContext.xml");
                Second s = (Second) ctx.getBean("two");
                System.out.println(s);
        }
}
```

Multiple Constructors: In a same class we can have multiple constructors. When ever we need to call multiple constructors through constructor injection then we need to configure the same class more than one time as shown in below example.

Example: Calling different constructors of the same class

//Bean.java

```
public class Bean {
        int sno;
        String sname;
        public Bean() {}
        public Bean(String sname) {
                this.sname = sname;
        }
        public Bean(int sno, String sname) {
                super();
                this.sno = sno;
                this.sname = sname;
        }
        public String toString() {
                return sno+" "+sname;
        }
}
```

```
// applicationContext.xml
<beans>
	<bean id = "one" class = "Bean">
		<constructor-arg>
			<value>Sai</value>
		</constructor-arg>
	</bean>
	<bean id = "two" class = "Bean">
		<constructor-arg>
			<value>1</value>
		</constructor-arg>
		<constructor-arg>
			<value>Kanakadhar</value>
		</constructor-arg>
	</bean>
</beans>
//Client.java
import org.springframework.context.support.ClassPathXmlApplicationContext;
public final class Client {
	public Client() {}
	public static void main(String[] args) {
		ClassPathXmlApplicationContext  ctx = new
	ClassPathXmlApplicationContext("applicationContext.xml");
		Bean b1 = (Bean) ctx.getBean("one"); //calls one parameter constructor
		Bean b2 = (Bean) ctx.getBean("two");//calls two parameterized constructor
		System.out.println(b1);
		System.out.println(b2);
	}
}
```

FactoryBean: BeanFactory:

1. A special bean class that can return other class object instead of its own object is called as FactoryBean.
2. In our regular programming we will come across many predefined FactoryBean classes.
3. When factory bean configuration bean id is configured as dependent value to our bean class property will not be injected with factory bean class object. It will be injected with some other class object returned by factory bean class.

4. To make spring bean class as factory bean class make that bean class to implement "org.springframework.beans.factory.FactoryBean" interface.

Example: To create our own FactoryBean class.

//Demo.java

```
public interface Demo {
	String hi();
}
```

//DemoBean.java

```
import java.util.Calendar;
import java.util.Date;
public class DemoBean implements Demo{
	private Date dt;
	private Calendar cl;
	public void setDt(Date dt)
	{
		this.dt = dt;
	}
	public void setCl(Calendar cl)
	{
		this.cl = cl;
	}
	public String hi()
	{
		return "date: "+dt+"   cl = "+cl;
	}
}
```

//TestBean.java [FactoryBean class]

```
import java.util.Calendar;
import org.springframework.beans.factory.FactoryBean;
public class TestBean implements FactoryBean{
	public Object getObject() throws Exception {
		System.out.println("getObject()");
		Calendar cl = Calendar.getInstance();
		return cl;
	}
```

```
2302        public Class getObjectType() {
2303                System.out.println("getObjectType()");
2304                return Calendar.class;
2305        }
2306        public boolean isSingleton() {
2307                System.out.println("isSingleton()");
2308                return true;
2309        }
2310  }
```

Note:TestBean class returns object of Calendar class.

```
2312  // applicationContext.xml
2313  <beans>
2314        <bean id = "dat" class = "java.util.Date"/>
2315        <bean id = "tbean" class = "TestBean"/>
2316        <bean id = "db" class = "DemoBean">
2317                <property name="dt">
2318                        <ref bean = "dat" />
2319                </property>
2320                <property name="cl">
2321                        <ref bean = "tbean"/>
2322                </property>
2323        </bean>
2324  </beans>
2325  //DemoClient.java
2326  import org.springframework.context.support.ClassPathXmlApplicationContext;
2327  public class   DemoClient
2328  {
2329      public static void main(String args[])
2330      {
2331          ClassPathXmlApplicationContext  ctx = new
2332  ClassPathXmlApplicationContext("applicationContext.xml");

2333          DemoBean d = (DemoBean) ctx.getBean("db");
2334          System.out.println(d.hi());
2335      }
```

}

Wiring: The process of configuring bean properties and performing dependency injection on bean is called as wiring operation.

There are two types of wiring operations:

1. Explicit wiring
2. Auto wiring

If spring bean properties are configured explicitly in spring configuration file for dependency injection is called as **explicit wiring**.

The dependency injection configurations that we have done so far in our programming come under explicit wiring.

If container is detecting dependant values to bean properties automatically without any configuration of properties, then it is called as **auto wiring.**

Limitations of auto wiring:

1. Only bean references can be injected. It means simple values of int, float etc.., can't be injected.
2. There is a possibility of ambiguity (confusion) problem. If multiple suitable dependent values are found for a bean property in auto wiring.

Spring allows auto wiring in 4 modes:

1. by name
2. by type
3. by constructor
4. by auto detect

To enable auto wiring on spring bean we need to use auto wire attribute of <bean> tag.

1. **byName:** This performs setter injection on bean property. The name of bean property and bean id of dependent bean class object must match by name.

Note: We should use auto wiring on bean property only when the bean property is ready to take another spring bean class object as a dependent value. No two bean tags should have same bean id. But same bean class can be configured with separate bean id's using separate bean tags.

2. **byType:** Performs setter injection on bean properties. In this mode of auto wiring the data type of bean properties must match with the dependent bean object data type. Here it doesn't checks whether the names were matched or not.

Note: No two properties should be configured with same data type. If it happens it will raise ambiguity error.

3. **constructor:** it performs dependency injection on bean properties by using the parameterized constructors [constructor injection]. Here also names need not to match but an appropriate parameterized constructor should be available.
4. **autodetect:** Chooses "constructor" or "byType"mode of auto wiring by scanning the details of spring bean class. If zero argument constructor is found inspring bean class then "byType" mode of auto wiring takes place. If appropriate and only parameterized constructor is available then "constructor" mode of auto wiring takes place.

Note: While working with pre defined spring bean classes and if we do not know its properties names or constructor details particularly then we can use this auto wiring concept.

Auto wiring is useful to work with predefined classes and third party API's.

In auto wiring and explicit wiring both were configured with same setter injection on single bean property with two different values then the explicit values will be affected.

Eg: byName & explicit.

If auto wiring is performing setter injection and explicit wiring is performing constructor injection on same bean properties with different values then auto wiring values will be affected.

If auto wiring performs constructor injection and explicit wiring is performing setter injection on the same property with different values then explicit values will be affected.

Note: We can also go for few properties with explicit wiring and few properties with autowiring.

//Demo.java

```
public interface Demo {
	String hi();
}
```

// DemoBean.java

```
public class DemoBean implements Demo{
	TestBean tb = null;
	public DemoBean()
	{
		System.out.println("Zero arg constructor of DemoBean class");
	}
	public DemoBean(TestBean tb)
	{
		System.out.println("One arg constructor of DemoBean class");
		this.tb = tb;
	}
	public void setTb(TestBean tb)
	{
		System.out.println("setTb() of DemoBean class");
		this.tb = tb;
	}
	public String hi()
	{
		return tb.toString();
	}
```

```
}
//TestBean.java
public class TestBean
{
	String msg;
	public TestBean() {
		System.out.println("zero arg. constructor of TestBean class.");
	}
	public void setMsg(String msg) {
		System.out.println("setMsg() of TestBean class");
		this.msg = msg;
	}
	@Override
	public String toString() {

		return msg;
	}
}
```

//**applicationContext.xml [Autowiring by name:** Here TestBean class "id" & DemoBean class property both are having same names. **]**

```
<beans>
	<bean id = "tb" class = "TestBean"> <!--  Explicit wiring -->
		<property name="msg">
			<value>Hi friends... Keep Rocking. :)</value>
		</property>
	</bean>
	<bean id = "db" class = "DemoBean" autowire = "byName"/>
</beans>
```

Note:

1. When we go for explicit wiring through setter injection then autowiring values will be overridden and explicit wiring values will be displayed.
2. When we go for explicit wiring through constructor injection then constructor injection values will be overridden by autowiring values

//DemoClient.java

```
import org.springframework.context.support.ClassPathXmlApplicationContext;
public class   DemoClient
{
      public static void main(String args[])
      {
         ClassPathXmlApplicationContext  ctx = new
ClassPathXmlApplicationContext("applicationContext.xml");
         DemoBean d = (DemoBean) ctx.getBean("db");
         System.out.println(d.hi());
      }
}
```

Exmple: Setter injection will always dominate Constructor injection

<!-- applicationContext.xml -->

```
<beans>
      <bean id = "tb" class = "TestBean"> <!--  Explicit wiring -->
            <property name="msg">
                  <value>Hi friends... Happy Republic Day.... :)</value>
            </property>
      </bean>
      <bean id = "tb1" class = "TestBean">
            <property name="msg">
                  <value>Gmg. friends.... :)</value>
            </property>
      </bean>
      <bean id = "db" class = "DemoBean" autowire = "byName"/>
            <constructor-arg>
                  <ref bean = "tb1"/>
            </constructor-arg>
      </bean>
</beans>
```

Note:

1. Here explicitly we are doing constructor injection.
2. First of all constructor values will be injected & again setter values will be injected.
3. So that setter values will be displayed & constructor values will be ignored.

//Example: Explicit injection dominates implicit injection when both are using either setter or constructor injection

```
<beans>
	<bean id = "tb" class = "TestBean"> <!--  Explicit wiring -->
		<property name="msg">
			<value>Hi friends... Happy Republic Day.... :)</value>
		</property>
	</bean>
	<bean id = "tb1" class = "TestBean">
		<property name="msg">
			<value>Gmg. friends.... :)</value>
		</property>
	</bean>
	<bean id = "db" class = "DemoBean" autowire = "byName"/>
		<property name="tb">
			<ref bean = "tb1"/>
		</property>
	</bean>
</beans>
```

Note:

1. In the above example auto wiring & explicit wiring both are trying to do setter injection. In this case explicit injection will dominate implicit injection.
2. Similarly if auto wiring & explicit wiring both are trying to do constructor injection, then also explicit injection will dominate implicit injection.

//Example: byType

Note: The entire example is same. The only modification is with configuration file.

<!-- applicationContext.xml -->

```
	<bean id = "t1" class = "TestBean"> <!--  Explicit wiring -->
		<property name="msg">
			<value>Gmg. friends... Have a joyful day. :)</value>
		</property>
	</bean>
	<bean id = "db" class = "DemoBean" autowire = "byType"/>
</beans>
```

Note:

1. Here TestBean class "id" & DemoBean class property both are different. But still here autowiring is working based up on data type of bean class property & Bean class.
2. In autowiring with byType the same class should not be configured more than once.

//Example: We should not do as follows: Because TestBean class is configured more than once where byType is used. If we configure more than once then it will throw amiguity error.

```
<!-- applicationContext.xml -->
<beans>
	<bean id = "t1" class = "TestBean"> <!--  Explicit wiring -->
		<property name="msg">
			<value>Gmg. friends... Have a joyful day. :)</value>
		</property>
	</bean>
	<bean id = "t2" class = "TestBean">
		<property name="msg">
			<value>Gmg.... friends Have a joyful day. :)</value>
		</property>
	</bean>
	<bean id = "db" class = "DemoBean" autowire = "byType"/>
</beans>
```

//Example: constructor

Note: The entire example is same. The only modification is with configuration file.

```
<!-- applicationContext.xml -->
<beans>
	<bean id = "t2" class = "TestBean"> <!--  Explicit wiring -->
		<property name="msg">
			<value>Gmg. friends... Have a joyful day. :)</value>
		</property>
	</bean>
	<bean id = "db" class = "DemoBean" autowire = "constructor"/>
</beans>
```

//Example: autodetect

<!-- applicationContext.xml -->

Note: The entire example is same. The only modification is with configuration file.

```
<beans>
	<bean id = "t1" class = "TestBean"> <!--  Explicit wiring -->
		<property name="msg">
			<value>Gmg. friends... Have a joyful day. :)</value>
		</property>
	</bean>
	<bean id = "db" class = "DemoBean" autowire = "autodetect"/>
</beans>
```

DAO Module:

Spring DAO:

Connection pooling:

Every jdbc data source object represents one jdbc connection pool. To access each connection object of connection pool we need to work with jdbc data source object.

Jdbc connection pool is a factory that contains set of readily available jdbc connection objects before actually being used.

In spring application we can work with 3 types of connection pools:

1. spring built in jdbc connection pool [org.sf.jdbc.datasource.DriverManagerDataSource class]

Note: This connection pool is not real jdbc connection pool. Because it does not pool jdbc connection objects. Moreover it creates one jdbc connection object dynamically based on demand. Using this style is not industrial standard.

2. Using third party connection pool software's. [c3p0, proxool, dc30, apache.. etc.,]
3. Using web/application server managed jdbc connection pools.

If our application is stand alone application or the application that runs outside the server [web/application server] then use third party connection pooling.

If our spring application is deployed in web/application server then use server managed connection pooling.

DAO class:

1. The java class or application that separates persistence logic from other applications is called as DAO.
2. DAO is very useful to make persistence logic as flexible logic for modification with out affecting the other logics of the application.
3. Spring DAO module is given to develop jdbc style persistence logic in any layer of the project by getting abstraction layer on jdbc programming.
4. Using spring DAO module we can develop jdbc style persistence logic without directly using jdbc logic and without performing common jdbc operations like establishing

connections, transaction management, jdbc operations, createing & closing of statement objects etc…

Plain jdbc code:

1. load or register jdbc driver
2. establish connection to database
3. create statement object
4. begin transaction
5. Perform persistent operations like insert, update, delete or read etc..,
6. gather results and process results
7. Commit or rollback based on transaction execution.
8. Close the connection

Spring DAO based persistence logic:

1. Get access to jdbc template class object through dependency injection process
2. perform persistence operations like insert, update, delete, read/select.
3. Gather results and process.

If we use jdbc template class given by spring dao module, it will take care of the common activities of jdbc programming like establishing connection, closing connection, transaction management & other common operations. So programmer can concentrate only on application specific operations.

Jdbc template class provides lots of methods to perform persistence operations on database s/w without using any kind of jdbc api.

query(_), queryForXxx(_) of jdbc template class are given to execute sql select queries.

update() of jdbc template class is given to execute non-select queries.

batchUpdate() is used for batch processing.

Limitations with jdbc:

1. All the steps are mandatory to follow.
2. ResultSet object is not serializable object so we cannot send it over network.
3. All jdbc exception are checked exceptions so programmer must catch and handle them.
4. Allows only positional parameters for queries (?).

Features of spring DAO:

1. Allows to develop jdbc style persistence logic without using jdbc api.
2. The jdbc template class takes care of the common jdbc workflow or operations. So programmer can concentrate only on application specific persistence logic development.
3. Select query results can be stored directly in collection framework data structures which are serializable objects by default.
4. Spring DAO internally uses jdbc code. So it catches all jdbc checked exceptions thrown by jdbc code and rethrows as unchecked exceptions.

5. Simplifies the process of developing jdbc style persistence logic by providing abstraction layer on jdbc programming.
6. Spring DAO based persistence logic is not database independent because it internally uses jdbc code and database specific queries given by programmer.
7. To execute select queries that returns multiple records use “queryForList()”.
8. To execute select query that returns a single record use “queryForMap()”.
9. To execute select query that contains aggregate function results or any individual values use “queryForLong()”, “queryForInt()” etc..,
10. Allows both positional (?) and named (<**:pname**> parameters for queries.
11. To work with named parameters we must use NamedParameterJdbcTemplate class.
12. Single connection data source represents a connection pool that contains only one jdbc connection object and will not be closed automatically when spring container is stopped.
13. In order to close this single connection we must call close() method implicitly. This class is subclass of DriverManageDataSource and both these classes related connection pools are not meant for real time programming.

Connection Pool:

1. Connection Pool is a pool of ready made specified number of database connection objects.
2. When ever client program needs a database connection they can request our pool and can get one connection object.
3. Client program can do all database operations and can release the connection object back to connection pool.
4. The same connection object can be reused any number of times till the connection pool is destroyed.
5. Connection objects will be in waiting state in connection pool for client request.
6. At a time only one client can use one connection object.
7. If all connection objects of connection pool are busy then client program should be waiting state till any client releases a connection object.
8. With connection pool we can increase performance of entire application.
9. In real time applications regularly we work with connection pool.
10. When huge amount of customers are trying to connect with database server in those cases better we use connection pooling concept.

Connection pool can be created in 3 ways:

1. Dummy Connection pool
2. Third Party Connection pool
3. Web server or application server supplied connection pool

Example: Integrating spring with Jdbc [Dummy Connection pool]

//SelectInter.java

```
public interface  SelectInter
{
	public String fetchEmpName(int eno);
	public long fetchEmpSalary(int eno);
}
```

//SelectImple.java

```
import java.sql.*;
import javax.sql.DataSource;

public class  SelectImple  implements SelectInter
{
	DataSource ds = null;
	public void setDs(DataSource ds)
	{
		this.ds = ds;
	}
	public String fetchEmpName(int eno)
	{
		try
		{
			//get access to jdbc connection object from jdbc connection pool through data source object "ds"
			Connection con = ds.getConnection();
			//persistence logic
			Statement st = con.createStatement();
			ResultSet rs = st.executeQuery("select ename from emp where empno = "+eno);

			if(rs.next())
			{
				return rs.getString(1);
			}
			else
			{
				System.out.println("No record found with employee number: "+eno);
			}

			rs.close();
			st.close();
			con.close();
		}
		catch(SQLException e)
		{
			e.printStackTrace();
			return "Record not found";
		}
		catch(Exception e)
```

```
                {
                        e.printStackTrace();
                        return "Record not found";
                }//catch
                return "Record not found";
        }//fetch empName
        public long fetchEmpSalary(int eno)
        {
                try
                {
                        //get access to jdbc connection object from jdbc connection pool through data
source object "ds"
                        Connection con = ds.getConnection();
                        //persistence logic
                        Statement st = con.createStatement();
                        ResultSet rs = st.executeQuery("select sal from emp where empno = "+eno);
                        if(rs.next())
                        {
                                return rs.getLong(1);
                        }
                        else
                        {
                                System.out.println("No record found with employee number: "+eno);
                        }
                        rs.close();
                        st.close();
                        con.close();
                }
                catch(SQLException e)
                {
                        e.printStackTrace();
                        return 0;
                }
                catch(Exception e)
                {
                        e.printStackTrace();
                        return 0;
                }//catch
                return 0;
        }//fetch empsalary
}
```

Note:

1. Its same as regular Jdbc program. But here we are getting Connection object by using DataSource object.
2. **DataSource** is an interface belongs to javax.sql package. The same interface has many implementation classes. One of the class is **DriverManagerDataSource**.

//spring.cfg.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC  "-//SPRING//DTD BEAN//EN"
```

```
"http://www.springframework.org/dtd/spring-beans.dtd">

<beans>
        <bean  id="dmds"
       class="org.springframework.jdbc.datasource.DriverManagerDataSource">

              <property  name="driverClassName">
                    <value>oracle.jdbc.driver.OracleDriver</value>
              </property>
              <property  name="url">
                    <value>jdbc:oracle:thin:@localhost:1521:sathya</value>
              </property>
              <property  name="username">
                    <value>scott</value>
              </property>
              <property  name="password">
                    <value>tiger</value>
              </property>
        </bean>
         <bean  id="sel"   class="SelectImple">

              <property  name="ds">
                    <ref bean = "dmds"/>
              </property>
         </bean>
</beans>
```

Note: In the above xml file we have given service id in url.
Q) How to find service id of oracle
Ans:
SQL> select * from global_name;
GLOBAL_NAME

SATYA.US.ORACLE.COM

//SelectClient.java

```
import org.springframework.core.io.FileSystemResource;
import org.springframework.beans.factory.xml.XmlBeanFactory;
public class   SelectClient
{
      public static void main(String args[])
      {

                  FileSystemResource  res=new FileSystemResource("spring.cfg.xml");
                  XmlBeanFactory  factory=new XmlBeanFactory(res);
                  SelectImple s=(SelectImple)factory.getBean("sel");
                  System.out.println("Employee name is: "+s.fetchEmpName(7839));
                  System.out.println("Employee salary is: "+s.fetchEmpSalary(7839));
      }
}
```

Example: Third Party Connection pool – Apache Tomcat supplied jar file

Note: Many third party connection pools are available. We'll see two among them.

1. Tomcat supplied connection pool [Need not to start tomcat server. Simply we need to **set the class path for "naming-factory-dbcp.jar" file]**
2. It is available in **Tomcat 5.5\common\lib\naming-factory-dbcp.jar**
3. Also set classpath to **classes12.jar**

Note: Only change is in configurationm file. The rest of resources will be same.

//spring.cfg.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC  "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
        <bean id="dbcpds"  class="org.apache.tomcat.dbcp.dbcp.BasicDataSource"
        destroy-method = "close">
                <property  name="driverClassName">
                        <value>oracle.jdbc.driver.OracleDriver</value>
                </property>
                <property  name="url">
                        <value>jdbc:oracle:thin:@localhost:1521:sathya</value>
                </property>
                <property  name="username">
                        <value>scott</value>
                </property>
                <property  name="password">
                        <value>tiger</value>
                </property>
                <property  name="initialSize">
                        <value>3</value>
                </property>
        </bean>
        <bean  id="sel"  class="SelectImple">
                <property  name="ds">
                        <ref bean = "dbcpds"/>
                </property>
        </bean>
    </beans>
```

Example: Third Party Connection pool – Hibernate supplied jar file

1. Tomcat supplied connection pool [Need not to start tomcat server. Simply we need to **set the class path for "c3p0-0.9.1.jar" file]**
2. It is available in **hibernate-home\lib\c3p0-0.9.1.jar**
3. Also set classpath to **classes12.jar**

Note: Only change is in configurationm file. The rest of resources will be same.

//spring.cfg.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC  "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
        <bean id="c3p0ds"  class="com.mchange.v2.c3p0.ComboPooledDataSource"
        destroy-method = "close">
                <property  name="driverClass">
                        <value>oracle.jdbc.driver.OracleDriver</value>
                </property>
                <property  name="jdbcUrl">
                        <value>jdbc:oracle:thin:@localhost:1521:sathya</value>
                </property>
                <property  name="user">
                        <value>scott</value>
                </property>
                <property  name="password">
                        <value>tiger</value>
                </property>
                <property  name="initialPoolSize">
                        <value>3</value>
                </property>
                <property  name="maxPoolSize">
                        <value>30</value>
                </property>
                <property  name="acquireIncrement">
                        <value>5</value>
                </property>
        </bean>
        <bean  id="sel"  class="SelectImple">
```

```
            <property  name="ds">
                  <ref bean = "c3p0ds"/>
            </property>
      </bean>
</beans>
```

Web or application server based connection pool:

1. The nick name or alias name given to the object that is registered with registry s/w is technically called as **JNDI name**.
2. Data source represents server managed connection pool.
3. To provide global visibility to this data source object from multiple client applications we need to register it with registry s/w having jndi name.
4. Client application always uses jdbc data source object to get access to connection object of connection pool.
5. Jdbc data source object is an object of a class which implements java.sql.DataSource interface.

Steps:

1. Create Jdbc connection pool for certain database s/w.
2. Create jdbc data source and link it with jdbc connection pool.
3. register jdbc data source with registry s/w for global visibility
4. Client application performs jndi lookup operation on registry & gets jdbc data source object.
5. getConnection() method on data source object gets one jdbc connection object from jdbc connection pool.
6. Client application uses this jdbc connection object for persistence operations on database.
7. con.close() releases jdbc connection object back to the connection pool so that the same object can be used by other clients.

Example: Application server supplied connection pool with Weblogic

set the class path for:

1. **"weblogic1.jar" file [C:\bea\weblogic81\server\lib\weblogic.jar]**
2. **classes12.jar**

Weblogic8.1 connection pooling:

Step1: Create user in weblogic & start admin server

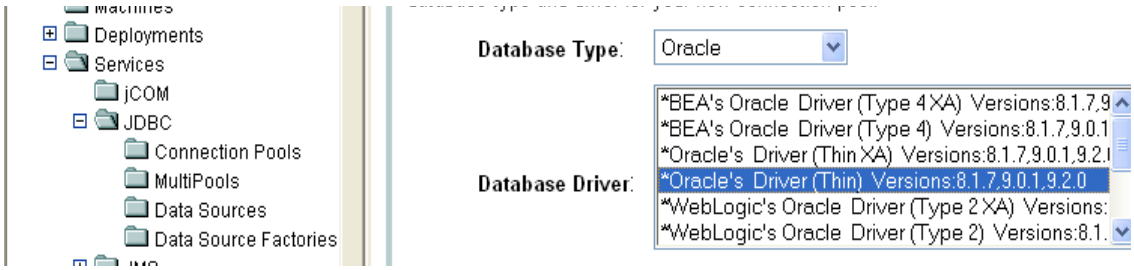
Start menu à programs à bea weblogic platform8.1 à configuration wizard à next à next à give password and confirm password as same [remind this password.] à next à next à specify configuration name [domain name] at right bottom of window à create à select start admin server à done.

Step2: open administrator console

Start menu à programs à bea weblogic platform8.1 à examples à weblogic workshop à server admin console à give password here à login.

Step3: Create Connection pool

Choose your domain name [in left pane of admin console window]à services à jdbc à click on **connection pools** à Configure a new JDBC Connection Pool à choose database and its corresponding driver as shown belowà continue



Define connection properties:

Name: give some name for connection pool

Database name: give service id if it is oracle

Host Name: ip address or server name where database server resides

Port: port no. of database server

Also provide database username & password and click on **continue** button à click on button "**test driver connection". à** if it shows connection successful then click on à "**Create & deploy".**

Step4: Create data source

Choose your domain name [in left pane of admin console window]à services à jdbc à click on **Data sources** à click on "Configure a new JDBC Data Source" à

Name: Give some name for Data source

JNDI name: Give some JNDI name. [remind this. We have to configure this in our program]

Select check box of : Emulate Two-Phase Commit for non-XA Driverà continue. à choose our connection pool from combo box à continue à create.

Note: Only change is in configurationm file. The rest of resources will be same.

//spring.cfg.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC  "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
        <bean  id="jofb"   class="org.springframework.jndi.JndiObjectFactoryBean">
                <property  name="jndiName">
                        <value>oraJNDI</value>
                </property>
                <property  name="jndiEnvironment">
                        <props>
        <prop key = "java.naming.factory.initial">weblogic.jndi.WLInitialContextFactory</prop>
```

```
                                <prop key = "provider_url">t3://localhost:7001</prop>
                        </props>
                </property>
        </bean>
        <bean  id="sel"   class="SelectImple">
                <property  name="ds">
                        <ref bean = "jofb"/>
                </property>
        </bean>
</beans>
```

Note:

1. As we have specified driver class, database url, user name & password in weblogic need not to specify once again in spring configuration file.
2. A connection pool is a factory that contains set of readily available jdbc connection objects before actually being used.
3. Jdbc DataSource object always represents one jdbc connection pool.
4. To access each conenction object of connection pool we need to use jdbc data source object.
5. org.springframework.jdbc.datasource.DriverManagerDataSource represents spring built in jdbc dummy connection pool by giving data source object.
6. This is called as dummy connection pool because it does not pools the connection objects. It creates connection object on demand.
7. DriverManagerDataSource implements javax.sql.DataSource interface.
8. This class uses given jdbc driver details and uses jdbc data source object which represents dummy connection pool.

   Note: Do not use **DriverManagerDataSource** in your real time projects.

Example: Working with JdbcTemplate class [queryForXxx() methods]

//SelectInter.java

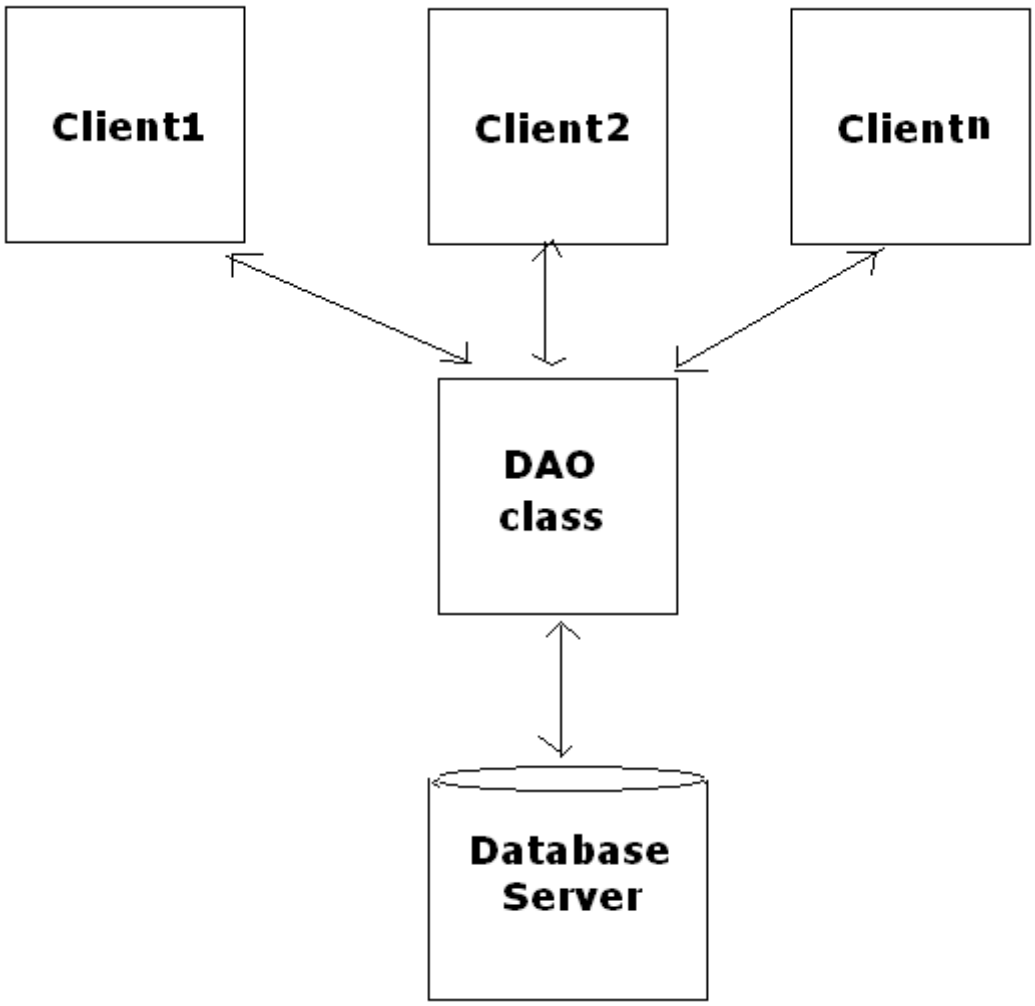
```
import java.util.List;
import java.util.Map;
public interface SelectInter {
        int getEmpCount(String desg);
        List getEmpDetails(String desg);
        Map getEmpDetails(int eno);
        boolean registerEmp(int no, String name, String desg, int sal);
        boolean modifyDesignation(int no, String newDesig);
        boolean fireEmp(int eno);
}
```

```
2971 //SelectImple.java
2972 import java.util.List;
2973 import java.util.Map;
2974 import org.springframework.jdbc.core.JdbcTemplate;
2975 public class SelectImple implements SelectInter{
2976                 JdbcTemplate jt;
2977                 String qry;
2978                 public void setJt(JdbcTemplate jt) {
2979                         this.jt = jt;
2980                 }
2981                 public boolean fireEmp(int eno) {
2982                         qry = "delete from emp where empno = ?";
2983                         int res = jt.update(qry, new Object[]{new Integer(eno)});

2984                         if(res == 0)
2985                                 return false;
2986                         else
2987                                 return true;
2988                 }
2989                 public int getEmpCount(String desg) {
2990                         qry = "select count(*) from emp where job = ?";
2991                         int count = jt.queryForInt(qry, new String[]{desg});
2992                         return count;
2993                 }
2994                 public Map getEmpDetails(int eno) {
2995                         qry = "select * from emp where empno = ? ";
2996                         Map m = jt.queryForMap(qry, new Object[]{new Integer(eno)});
2997                         return m;
2998                 }
2999                 public List getEmpDetails(String desg) {
3000                         qry = "select * from emp where job = ?";
3001                         List l = jt.queryForList(qry, new Object[]{desg});
3002                         return l;
3003                 }
3004                 public boolean modifyDesignation(int no, String newDesig) {
```

```
3005                    qry = "update emp set job = ? where empno = ?";
3006                    int res = jt.update(qry, new Object[]{newDesig, new Integer(no)});
3007                    if(res == 0)
3008                            return false;
3009                    else
3010                            return true;
3011            }
3012            public boolean registerEmp(int no, String name, String desg, int sal) {
3013                    qry = "insert into emp  (empno, ename, job, sal) values(?, ?, ?, ?)";
3014                    int res = jt.update(qry, new Object[]{new Integer(no), name, desg, new
3015 Integer(sal)});
3016                    if(res == 0)
3017                            return false;
3018                    else
3019                            return true;
3020            }
3021 }
3022 //applicationContext.xml
3023 <?xml version="1.0" encoding="UTF-8"?>
3024 <beans
3025         xmlns="http://www.springframework.org/schema/beans"
3026         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3027         xsi:schemaLocation="http://www.springframework.org/schema/beans
3028 http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">

3029 <!--  note: add naming-factory-dbcp.jar file from tomcat-home\common\lib  -->
3030          <bean  id="dbcp"   class="org.apache.tomcat.dbcp.dbcp.BasicDataSource">
3031                  <property  name="driverClassName">
3032                          <value>oracle.jdbc.driver.OracleDriver</value>
3033                  </property>
3034                  <property  name="url">
3035                          <value>jdbc:oracle:thin:@localhost:1521:sathya</value>
3036                  </property>
3037                  <property  name="username">
3038                          <value>scott</value>
3039                  </property>
```

```
3040                    <property  name="password">
3041                            <value>tiger</value>
3042                    </property>
3043                    <property  name="initialSize">
3044                            <value>3</value>
3045                    </property>
3046            </bean>
3047            <bean  id="template"   class="org.springframework.jdbc.core.JdbcTemplate">
3048                    <property  name="dataSource">
3049                            <ref bean = "dbcp"/>
3050                    </property>
3051            </bean>
3052            <bean  id="sel"   class="SelectImple">
3053                    <property  name="jt">
3054                            <ref bean = "template"/>
3055                    </property>
3056      </bean>
3057  </beans>
3058  //SelectClient.java
3059  import java.util.List;
3060  import java.util.Map;
3061  import org.springframework.context.support.ClassPathXmlApplicationContext;
3062  public class   SelectClient
3063  {
3064          public static void main(String args[])throws Exception
3065          {
3066                          ClassPathXmlApplicationContext ctx = new
3067  ClassPathXmlApplicationContext("applicationContext.xml");
3068                          SelectImple s=(SelectImple)ctx.getBean("sel");
3069                          System.out.println("Employee count with Clerk disgnation is:
3070  "+s.getEmpCount("CLERK"));
3071                          Map m = s.getEmpDetails(7934);
3072                          System.out.println("Details of empno: 7934 are: "+m.toString());
3073                          System.out.println("Clerk designation employees details are: ");
3074                          List l = s.getEmpDetails("CLERK");
3075                          for(int i = 0; i < l.size(); i++)
```

```
3076                    {
3077                        Map m1 = (Map) l.get(i);
3078                        System.out.println(m1.toString());
3079                    }
3080                    boolean bool = s.registerEmp(1, "Kanakadhar", "MANAGER", 50000);
3081                    System.out.println("Employee registered "+bool);
3082                    bool = s.modifyDesignation(1, "PRESIDENT");
3083                    System.out.println("Employee designation updated "+bool);
3084                    Thread.sleep(10000);
3085                    bool = s.fireEmp(1);
3086                    System.out.println("Employee fired "+bool);
3087        }
3088    }
```



3089 **Note:** The same DAO class can be shared by multiple client programs. But in our example we have
3090 seen only one client program which is calling all methods of DAO.

Example: Named parameter without container

```
import org.springframework.jdbc.core.namedparam.MapSqlParameterSource;
import org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate;
import org.springframework.jdbc.datasource.SingleConnectionDataSource;
public class Client {
	public static void main(String[] args) {
		SingleConnectionDataSource ds = new SingleConnectionDataSource();
		ds.setDriverClassName("oracle.jdbc.driver.OracleDriver");
		ds.setUrl("jdbc:oracle:thin:@localhost:1521:sathya");
		ds.setUsername("scott");
		ds.setPassword("tiger");
		NamedParameterJdbcTemplate template = new NamedParameterJdbcTemplate(
		String qry = "select count (*) from emp where job = :desig";
		MapSqlParameterSource params = new MapSqlParameterSource();
		params.addValue("desig", "CLERK");
		int count = template.queryForInt(qry, params);
		System.out.println("No. of Clerks are: "+count);
		ds.destroy();
	}
}
```

//Example: Named parameter with container

// SelectInter.java

```
public interface SelectInter {
	int getEmpCount(int eno1, int eno2);
}
```

//SelectImple.java

```
import org.springframework.jdbc.core.namedparam.MapSqlParameterSource;
import org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate;
//Add apache tomcat third party jar "naming-factory-dbcp.jar" file to build path
//Add classes12.jar for build path
public class SelectImple implements SelectInter{
		NamedParameterJdbcTemplate nt;
		String qry = "select count(*) from emp where empno >= :e1 and empno <= :e2";
		public void setNt(NamedParameterJdbcTemplate nt) {
			this.nt = nt;
```

```
3126                }
3127                public int getEmpCount(int eno1, int eno2) {
3128                        MapSqlParameterSource params = new MapSqlParameterSource();
3129                        params.addValue("e1", new Integer(eno1));
3130                        params.addValue("e2", new Integer(eno2));
3131                        int count = nt.queryForInt(qry, params);
3132                        return count;
3133                }
3134 }
3135 // applicationContext.xml
3136 <!-- note: add naming-factory-dbcp.jar file from tomcat-home\common\lib -->
3137 <beans>
3138         <bean id="dbcp" class="org.apache.tomcat.dbcp.dbcp.BasicDataSource">
3139                 <property name="driverClassName">
3140                         <value>oracle.jdbc.driver.OracleDriver</value>
3141                 </property>
3142                 <property name="url">
3143                         <value>jdbc:oracle:thin:@localhost:1521:sathya</value>
3144                 </property>
3145                 <property name="username">
3146                         <value>scott</value>
3147                 </property>
3148                 <property name="password">
3149                         <value>tiger</value>
3150                 </property>
3151                 <property name="initialSize">
3152                         <value>3</value>
3153                 </property>
3154     </bean>
3155     <bean id="namedTemplate"
3156         class="org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate"
3157          <constructor-arg>
3158                         <ref bean = "dbcp"/>
3159          </constructor-arg>
3160     </bean>
```

```
<bean id="sel" class="SelectImple">
        <property name="nt">
            <ref bean = "namedTemplate"/>
        </property>
    </bean>
</beans>
```

//SelectClient.java

```
import org.springframework.context.support.ClassPathXmlApplicationContext;
public class SelectClient
{
    public static void main(String args[])
    {
            ClassPathXmlApplicationContext ctx = new
ClassPathXmlApplicationContext("applicationContext.xml");
            SelectImple s=(SelectImple)ctx.getBean("sel");
            System.out.println("Employees in between empid 7500 to 7800 count is:
"+s.getEmpCount(7500, 7800));
    }
}
```

Example: To call oracle stored procedure

Note:

1. Create this procedure through sql> prompt of oracle client.
2. Make sure that "emp" table is esisiting in oracle. Normally it is a default table in scott user.

```
CREATE OR REPLACE PROCEDURE GET_EMP_DATA
      (NAME IN VARCHAR,
       NO OUT NUMBER,
        DESG OUT VARCHAR,
       SALARY OUT NUMBER)
AS
BEGIN
    SELECT EMPNO, JOB, SAL
        INTO NO, DESG, SALARY
        FROM EMP
        WHERE ENAME = NAME;
END;
```

Employee.java [POJO class]

```
class Employee {
	private Integer no;
	private String name;
	private String designation;
	private Double salary;
	public void setNo(Integer no) {
		this.no = no;
	}
	public void setName(String name) {
		this.name = name;
	}
	public void setDesignation(String designation) {
		this.designation = designation;
	}
	public void setSalary(Double salary) {
		this.salary = salary;
	}
	public String toString() {
		return this.no + "\t"
			+ this.name + "\t"
			+ this.designation + "\t"
			+ this.salary;
	}
}
```

// SimpleJdbcCallTest.java

```
/*
steps:
1. create datasource
2. pass datasource to subclass of StoredProcedure which is a static innerclass
3. Register all in & out parameters
4. read data from Map object, set to bean class object & display
*/
import java.sql.Types;
import java.util.HashMap;
```

```
3231 import java.util.Map;
3232 import javax.sql.DataSource;
3233 import org.springframework.jdbc.core.SqlOutParameter;
3234 import org.springframework.jdbc.core.SqlParameter;
3235 import org.springframework.jdbc.object.StoredProcedure;
3236 import org.springframework.jdbc.datasource.*;
3237 public class SimpleJdbcCallTest {
3238    public static void main(String args[]) {
3239    DriverManagerDataSource dataSource = new DriverManagerDataSource();
3240          dataSource.setDriverClassName("oracle.jdbc.driver.OracleDriver");
3241          dataSource.setUrl("jdbc:oracle:thin:@localhost:1521:SATHYA");
3242          dataSource.setUsername("scott");
3243          dataSource.setPassword("tiger");
3244          Employee emp = (Employee)
3245                      getEmployeeDetailsWithStoredProcedure(dataSource,"ALLEN");
3246          System.out.println("Allen Details : " + emp);
3247    }

3248  private static Object getEmployeeDetailsWithStoredProcedure
3249                                          (DataSource ds, String ename) {
3250          MyStoredProcedure sp = new MyStoredProcedure(ds);
3251          //call procedure
3252          Map results = sp.myexecute(ename);
3253          //set outparmeter values to emp object
3254          Employee emp = new Employee();
3255          emp.setName(ename);
3256          emp.setNo((Integer)results.get("NO"));
3257          emp.setDesignation((String)results.get("DESG"));
3258          emp.setSalary((Double)results.get("SALARY"));
3259          return emp;
3260  }//method2
3261 //MyStoredProcedure.java
3262  private static class MyStoredProcedure extends StoredProcedure {
3263          public MyStoredProcedure(DataSource ds) {
3264                  super(ds, "GET_EMP_DATA");
```

```
                this.setFunction(false); //false -> indicates its a stored procedure
                SqlParameter[] params = {
                        new SqlParameter("NAME", Types.VARCHAR),
                        new SqlOutParameter("NO", Types.INTEGER),
                        new SqlOutParameter("DESG", Types.VARCHAR),
                        new SqlOutParameter("SALARY", Types.DOUBLE)
                                        };
                this.setParameters(params);
                compile();
        }//constructor
        public Map myexecute(String name) {
                HashMap map = new HashMap();
                map.put("NAME",name);
                return super.execute(map);
        }
   }//inner class
}//outerclass
```

ORM Module:

1. Spring doesn't provide any of its own ORM tool. Simply it provides us a chance to integrate with any orm tool.
2. We can integrate spring with any ORM module in two ways:
    1. **Plain:** In plain style almost complete code should be written by programmer.
    2. **Template class:** Using template class we can foucs only on database operations instead of focusing on common code.

Note: We will see how to integrate spring & hibernate which is a ORM tool. So before we see this lets see a pure example regarding hibernate. So we can feel comfortable ragrding integration of spring n hibernate.

Example: Insertion of a record into employee table of oracle database using java hibernate application.

Files to be created:

1. hibernate.cfg.xml [Hibernate configuration file]
2. Employee.java [persistent class/POJO]
3. Employee.hbm.xml [Hibernate Mapping file]
4. Client.java [Java application using hibernate API & setup to interact with Database software.]

Step1: Create table in oracle database software as follows:

```
create table employee
(
eid number primary key,
firstname varchar2(20),
lastname varchar2(20),
email varchar2(20)
);
```

Note: Whether we configure primary key while creating table or not, but its compulsory to have primary key configuration in mapping file.

Step2: Create hibernate configuration file as follows:[Note: For reference open file from Hibernate-home\etc\hibernate.cfg.xml

Note: Here **Hibernate-home** indicates hibernate setup.[that is extracted folder of hibernate zip file which is a installation process of hibernate software.]

hibernate.cfg.xml

```
<!DOCTYPE hibernate-configuration PUBLIC
 "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
 "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
 <session-factory>
 <property name="hibernate.connection.driver_class">oracle.jdbc.driver.OracleDriver</property>
<property name="hibernate.connection.url">jdbc:oracle:thin:@localhost:1521:sathya</property>
<property name="hibernate.connection.username">scott</property>
<property name="hibernate.connection.password">tiger</property>
<property name="hibernate.dialect">org.hibernate.dialect.Oracle9Dialect</property>
<property name="show_sql">true</property>
<mapping resource="Employee.hbm.xml"/>
</session-factory>
</hibernate-configuration>
```

Note: Hibernate property tag names are fixed but their values will be changed based on version & name of the Database software. Refer **Hibernate-home\etc\hibernate.properties** file to get each database specific Hibernate properties values.

Step3: Develop POJO class/persistent class

```
3333 //Employee.java
3334 public class Employee implements java.io.Serializable
3335 {
3336  int no;
3337 String fname, lname, email;
3338  public void setNo(int no)
3339 {
3340 this.no = no;
3341 }
3342 public int getNo()
3343 {
3344  return no;
3345 }
3346 public void setFname(String fname)
3347 {
3348  this.fname = fname;
3349 }
3350 public String getFname()
3351 {
3352 return fname;
3353 }
3354 public void setLname(String lname)
3355 {
3356  this.lname = lname;
3357 }
3358 public String getLname(){
3359 return lname;
3360 }
3361 public void setEmail(String email){
3362 this.email = email;
3363 }
3364 public String getEmail(){
3365 return email;
3366 }
3367 }//class
```

Step 4: Develop Hibernate Mapping file (Employee.hbm.xml)

Note: Take reference file from **Hibernate-home\eg*.hbm.xml**

Employee.hbm.xml

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
        <class name="Employee" table="employee">
                <id name="no" column = "eid"/> <!-- Singular primary key field -->
                <property name="fname"  column="firstname"/>
                <property name="lname" column = "lastname"/>
                <property name = "email" column = "email"/>
        </class>
</hibernate-mapping>
```

Note: To see these SQL queries as log messages use **hibernate.show_sql** property in hibernate configuration file as shown below.

```
<property name = “hibernate.show_sql”>true</property>
```

Step 5: Develop hibernate based java application with persistence logic operations. [Insertion of record]

Client.java

```
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;
import org.hibernate.Transaction;
public class TestClient
{
public static void main(String[] args) throws Exception
{
        //locate and read configuration file
        Configuration conf = new Configuration();
        conf = conf.configure(); //if we name configuration file differently then we need to specify it as a argument
         //conf = conf.configure(“oracle.cfg.xml”);
        //creation of session factory object based on configuration file details
        SessionFactory factory = conf.buildSessionFactory();
```

```
        //get session object
        Session ses = factory.openSession();
       Transaction tx = ses.beginTransaction();
       //object creation with data
       Employee e1 = new Employee(); //Transient state
        e1.setNo(1);
       e1.setFname("Vara");
       e1.setLname("Prasad");
       e1.setEmail("kanakavaraprasad@gmail.com");
        //Note: Even after storing data, the object "e1" is in transient state.
       //record insertion
        ses.save(e1); //now the object "e1" is in persistent state.
        tx.commit();
       ses.close(); //closes statement and connection objects
        //Note: Now the object "e1" is in transient state.
       factory.close(); //destroys entire connection pool
    }
}
```

Points to ponder:

- conf.configure() method takes "**hibernate.cfg.xml"** file as a default configuration file. If you change the configuration file name as any other name then we need to pass its name as a string argument to the method configure().
- Non select operation must be executed as Transactional operations in hibernate based application. If not no error arises but the updation you have to database wont be affected. So mind it.
- After completion of task with database software close database connection without fail using ses.close(); If not no error will arise but object will be engaged as a busy connection till end of the particular client application.
- ses.save() method inserts record into the database table returns primary key value in the form of object and makes POJO class object to represent inserted record with synchronization between them.
- Every hibernate session object contains buffer. In order to flush this buffer use ses.flush() method.
- Hibernate software generates database specific sql queries internally for the methods we call from hibernate software to do persistent operations.

Step 6: Add the following jar files to class path to recognize hibernate API which is a third party API. They are as follows:

1.Main Jar file

1. hibernate3.jar [available in hibernate-home directory]

2.**Dependent jar files of "hibernate3.jar"** [available in hibernate-home\lib folder]

1. dom4j-version.jar
2. cglib-version.jar
3. commons-collections-version.jar
4. commons-logging-version.jar
5. jta.jar
6. asm.jar
7. antlr-version.jar

classes12.jar [to support oracle type 4 driver for oracle9i] (available in **oracle-home\ora92\jdbc\lib**)

Note:

- To get information about required jar files refer _readme.txt file in **hibernate-home\lib** folder
- When java application uses third party API then the API related jar files must be added in the class path.
- It is always recommended to add jar files in my computer environmental variables class path.

Step 7: Compile *.java files using javac tool.

Step 8: Execute client application [Client]

Example: Plain integeration of spring & Hibernate

// DemoInter.java

```
import java.util.Iterator;
public interface DemoInter {
	public Iterator getData() throws Exception;
}
```

// DemoImpl.java

```
import java.util.Iterator;
import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
public class DemoImpl implements DemoInter{
	private SessionFactory factory = null;
	public void setFactory(SessionFactory factory) {
		System.out.println("Setter of spring class");
		this.factory = factory;
```

```
3478        }
3479        public Iterator getData() throws Exception {
3480            System.out.println("In spring class getData()");
3481            Session  ses = factory.openSession();
3482            Query query = ses.createQuery("from User");
3483            Iterator i1 = query.iterate();
3484            //ses.close();
3485            return i1;
3486        }
3487    }
```

Table creation in oracle:

```
3488    create table users
3489    (
3490        userid number(5) primary key,
3491        uname varchar2(20),
3492        role varchar2(20)
3493    )
```

//User.java

```
3494    public class User {
3495        private int uid;
3496        private String uname, role;
3497        public String getUname() {
3498            return uname;
3499        }
3500        public void setUname(String uname) {
3501            this.uname = uname;
3502        }
3503        public String getRole() {
3504            return role;
3505        }
3506        public void setRole(String role) {
3507            this.role = role;
3508        }
3509        public int getUid() {
3510            return uid;
3511        }
3512
```

```
3513            }
3514            public void setUid(int uid)
3515            {
3516                    this.uid = uid;
3517            }
3518    }
3519    // User.hbm.xml
3520    <?xml version="1.0" encoding="utf-8"?>
3521    <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
3522    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
3523    <hibernate-mapping>
3524       <class name="User" table="users" schema="SCOTT">
3525          <id name="uid" column = "userid"/>
3526          <property name="uname"/>
3527          <property name="role"/>
3528             </class>
3529    </hibernate-mapping>
3530    // SpringHibernate.xml
3531    <?xml version="1.0" encoding="UTF-8"?>
3532    <beans
3533            xmlns="http://www.springframework.org/schema/beans"
3534            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3535            xsi:schemaLocation="http://www.springframework.org/schema/beans
3536    http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">

3537            <bean id = "ds" class =
3538    "org.springframework.jdbc.datasource.DriverManagerDataSource">
3539                    <property name="driverClassName">
3540                            <value>oracle.jdbc.driver.OracleDriver</value>
3541                    </property>
3542                    <property name="url">
3543                    <value>jdbc:oracle:thin:@localhost:1521:sathya</value>
3544                    </property>
3545                    <property name="username">
3546                            <value>scott</value>
3547                    </property>
```

```
<property name="password">
<value>tiger</value>
</property>
</bean>
<bean id = "mySessionFactory" class =
"org.springframework.orm.hibernate3.LocalSessionFactoryBean">
<property name="dataSource" ref = "ds"/>
<property name="mappingResources">
<list>
<value>User.hbm.xml</value>
</list>
</property>
<property name="hibernateProperties">
<props>
<prop key="hibernate.dialect">org.hibernate.dialect.Oracle9Dialect</prop>
<prop key="show_sql">true</prop>
</props>
</property>
</bean>
<bean id = "d1" class = "DemoImpl">
<property name="factory" ref = "mySessionFactory" />
</bean>
</beans>
```

Note:

1. We can configure both spring & hibernate properties inside spring configuration file as shown above.
2. In above xml file we have specified **show_sql** as true. But still it work incase of spring. If we use a separate xml file for hibernate configuration then it supports. We'll see its example later.

//Client.java

```
import java.util.Iterator;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
public class Client {
public static void main(String[] args) throws Exception {
ApplicationContext ctx = new
```

```
ClassPathXmlApplicationContext("SpringHibernate.xml");
                BeanFactory factory = (BeanFactory)ctx;
                DemoInter d = (DemoInter)factory.getBean("d1");
                Iterator i1 = d.getData();
                while(i1.hasNext())
                {
                        User u1 = (User)i1.next();
                        System.out.println(u1.getUid()+" "+u1.getUname()+" "+u1.getRole());
                }
        }
}
```

//Example: Integration of spring & Hibernate using HiberanteTemplate class

Note: Interface, pojo class, table, mapping file & client program are same as above example.

//DemoImpl.java

```
import java.util.Iterator;
import java.util.List;
import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.springframework.orm.hibernate3.HibernateTemplate;
public class DemoImpl implements DemoInter{
        private HibernateTemplate ht = null;
        public void setHt(HibernateTemplate ht) {
                System.out.println("setter of spring");
                this.ht = ht;
        }
        public Iterator getData() throws Exception {
                System.out.println("In spring class getData()");
                List l = ht.find("from User");
                Iterator i1 = l.iterator();
                return i1;
        }
}
```

```
<!--SpringHibernate.xml-->
<?xml version="1.0" encoding="UTF-8"?>
<beans
	xmlns="http://www.springframework.org/schema/beans"
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">
	<bean id = "ds" class =
	"org.springframework.jdbc.datasource.DriverManagerDataSource">
		<property name="driverClassName">
			<value>oracle.jdbc.driver.OracleDriver</value>
		</property>
		<property name="url">
		<value>jdbc:oracle:thin:@localhost:1521:sathya</value>
		</property>
		<property name="username">
			<value>scott</value>
		</property>
		<property name="password">
		<value>tiger</value>
		</property>
	</bean>
	<bean id = "mySessionFactory" class =
"org.springframework.orm.hibernate3.LocalSessionFactoryBean">
		<property name="dataSource" ref = "ds"/>
		<property name="mappingResources">
			<list>
				<value>User.hbm.xml</value>
			</list>
		</property>
	<property name="hibernateProperties">
		<props>
			<prop key="hibernate.dialect">org.hibernate.dialect.Oracle9Dialect</prop>
			<prop key="show_sql">true</prop>
		</props>
	</property>
```

```
        </bean>
        <bean id = "template" class = "org.springframework.orm.hibernate3.HibernateTemplate">
                <constructor-arg>
                        <ref bean = "mySessionFactory"/>
                </constructor-arg>
        </bean>
        <bean id = "d1" class = "DemoImpl">
                <property name="ht" ref = "template"/>
        </bean>
</beans>
```

Example: Named Query

Note: Instead of writing query in client program we can write it inside mapping file. So the same query can be used by multiple files & maintenance will becomes easy.

Note: Table, Pojo class, Sping n hibernate configuration file are same as above example

```
<!--User.hbm.xml-->
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
        <class name="User" table="users" schema="SCOTT">
          <id name="uid" column = "userid"/>
          <property name="uname"/>
          <property name="role"/>
        </class>
        <query name = "users">
                from User where uname like ?    <!--hql query-->
        </query>
</hibernate-mapping>
//DemoClient.java
import java.util.Iterator;
import java.util.List;
import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
```

```
import org.springframework.orm.hibernate3.HibernateTemplate;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
public class DemoClient
{
	private HibernateTemplate ht = null;
	public void setHt(HibernateTemplate ht) {
		System.out.println("setter of spring");
		this.ht = ht;
	}
	public static void main(String[] args) throws Exception {
		ApplicationContext ctx = new
ClassPathXmlApplicationContext("SpringHibernate.xml");
		BeanFactory factory = (BeanFactory)ctx;
		DemoClient d = (DemoClient)factory.getBean("d1");
		List l =d.ht.findByNamedQuery("users", new Object[]{"%S%"});
		Iterator it=l.iterator();
		while(it.hasNext())
		{
			User ob=(User)it.next();
			System.out.print(ob.getUid());
			System.out.println(" "+ob.getUname()+" "+ob.getRole());
			System.out.println("--------------");
		}
	}
}
```

Example: Separate configuration files for Spring & Hibernate

```
<!--spring.cfg.xml-->
<?xml version="1.0" encoding="UTF-8"?>
<beans
	xmlns="http://www.springframework.org/schema/beans"
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">
```

```
3723        <bean id = "ds" class = "org.springframework.jdbc.datasource.DriverManagerDataSource">
3724                <property name="driverClassName">
3725                        <value>oracle.jdbc.driver.OracleDriver</value>
3726                </property>
3727                <property name="url">
3728                <value>jdbc:oracle:thin:@localhost:1521:sathya</value>
3729                </property>
3730                <property name="username">
3731                        <value>scott</value>
3732                </property>
3733                <property name="password">
3734                <value>tiger</value>
3735                </property>
3736        </bean>
3737        <bean id = "mySessionFactory" class =
3738 "org.springframework.orm.hibernate3.LocalSessionFactoryBean">
3739                <property name="dataSource" ref = "ds"/>
3740                <property name = "configLocation">
3741                        <value>hibernate.cfg.xml</value>
3742                </property>
3743        </bean>
3744        <bean id = "template" class = "org.springframework.orm.hibernate3.HibernateTemplate">
3745                <constructor-arg>
3746                        <ref bean = "mySessionFactory"/>
3747                </constructor-arg>
3748        </bean>
3749        <bean id = "d1" class = "DemoClient">
3750                <property name="ht" ref = "template"/>
3751        </bean>
3752 </beans>
3753 <!--hibernate.cfg.xml-->
3754 <!DOCTYPE hibernate-configuration PUBLIC
3755         "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
3756         "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
```

```
<hibernate-configuration>
        <session-factory>
            <property name = "hibernate.dialect">org.hibernate.dialect.Oracle9Dialect</property>
            <property name="show_sql">true</property>
            <mapping resource="User.hbm.xml"/>
        </session-factory>
</hibernate-configuration>
```

Note: The rest of resources are same as earlier examples.

Example: Updating multiple records using HibernateTempalte class

//DemoClient.java

```
import java.util.Iterator;
import java.util.List;
import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.springframework.orm.hibernate3.HibernateTemplate;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
public class DemoClient{
        private HibernateTemplate ht = null;
        public void setHt(HibernateTemplate ht) {
                System.out.println("setter of spring");
                this.ht = ht;
        }
        public static void main(String[] args) throws Exception {
                ApplicationContext ctx = new
        ClassPathXmlApplicationContext("SpringHibernate.xml");
                BeanFactory factory = (BeanFactory)ctx;
                DemoClient d = (DemoClient)factory.getBean("d1");
                String qry = "update User set role = ? where role = ?";
                int res = d.ht.bulkUpdate(qry, new Object[]{"PM", "SSE"});
                System.out.println("No. of records updated are: "+res);
        }
}
```

Note: The rest of resources are same as earlier example.

Example: Retrieving all records of a table using HibernateTemplate class

Table in oracle:

```
create table event
(
        eid number(5) primary key,
        evname varchar2(25)
)
```

//eventpojo.java

```
public class eventpojo
{
        Integer eventid;
        String eventname;
        public eventpojo(){ }
        public Integer getEventid() {
                return eventid;
        }
        public void setEventid(Integer eventid) {
                this.eventid = eventid;
        }
        public String getEventname() {
                return eventname;
        }
        public void setEventname(String eventname)
        {
                this.eventname = eventname;
        }
}
```

<!-- eventpojo.hbm.xml -->

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC  "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
   "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
```

```
<hibernate-mapping>
	<class name="eventpojo" table="event">
		<id  name="eventid" column="eid"/>
		<property name="eventname" column="evname"/>
	</class>
</hibernate-mapping>
```

//eventdao.java

```
import org.springframework.orm.hibernate3.*;
import java.util.*;
public class eventdao
{
	HibernateTemplate ht;
	public void setHt(HibernateTemplate ht)
	{
		this.ht=ht;
	}
	public void saveObject(Object o)
	{
		ht.save(o);
	}
	public void selectAll()
	{
		List  l =ht.loadAll(eventpojo.class);
		Iterator it=l.iterator();
		while(it.hasNext())
		{
			eventpojo ob=(eventpojo)it.next();
			System.out.print(ob.getEventid());
			System.out.println(" "+ob.getEventname());
			System.out.println("--------------");
		}
	}
}
```

```
<!-- spring.cfg.xml -->
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC  "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
  <bean id="id1" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
   <property name="driverClassName">
    <value>oracle.jdbc.driver.OracleDriver</value>
   </property>
   <property name="url">
   <value>jdbc:oracle:thin:@localhost:1521:sathya</value>
   </property>
   <property name="username">
   <value>scott</value>
   </property>
   <property name="password">
   <value>tiger</value>
   </property>
  </bean>
  <bean id="id2" class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
  <property name="dataSource">
    <ref bean="id1"/>
   </property>
  <property name="mappingResources">
   <list>
    <value> eventpojo.hbm.xml </value>
   </list>
  </property>
  <property name="hibernateProperties">
   <props>
    <prop key="hibernate.dialect">org.hibernate.dialect.Oracle9Dialect </prop>
    <prop key="show_sql"> true </prop>
   </props>
   </property>
  </bean>
```

```
    <bean id="id3" class="org.springframework.orm.hibernate3.HibernateTemplate">
     <property name="sessionFactory">
        <ref bean="id2"/>
     </property>
    </bean>
    <bean id="id4" class="eventdao">
    <property name="ht">
     <ref bean="id3"/>
    </property>
    </bean>
</beans>
```

//client.java

```
import org.springframework.core.io.*;
import org.springframework.beans.factory.*;
import org.springframework.beans.factory.xml.*;
public class client
{
        public static void main(String[] args)
        {
                BeanFactory factory=new XmlBeanFactory(new
        ClassPathResource("spring.cfg.xml"));
                eventdao ed =(eventdao)factory.getBean("id4");
                eventpojo  ep=new eventpojo();
                ep.setEventid(new Integer(2));
                ep.setEventname("WindowEvent");
                ed.saveObject(ep);
                ed.selectAll();
                System.out.println("completed...");
        }
}
```

JEE Module:

Example: TimerTask [Job Scheduling] When ever we need to execute any task at particular time either for one time or for every particular period then we can use scheduling as follows:

//task1.java [Core java program]

```
import java.util.*;
```

```
public class  task1  extends TimerTask
{
   public void run()
   {
        System.out.println("welcome to sathya ");
    }
}
```

//scheduledemo.java

```
import java.util.*;
public class scheduledemo
{
      public static void main(String args[])
      {
        Timer t1=new Timer();
        t1.schedule(new task1(),5000); //initial delay
        t1.schedule(new task1(),5000,3000);  //initial delay, periodical delay
        }
}
```

Example: Multiple Tasks

Note: "task1" class is same as above.

//task2 .java

```
import java.util.*;
import java.io.*;
public class  task2  extends TimerTask
{
   public void run()    {
        try
        {
        FileWriter  fp=new FileWriter("demofile.txt", true);
        fp.write(new Date().toString()+"\n");
        fp.close();
        System.out.println("data stored");
        }catch(Exception e){}
    }
}
```

```
<!-- Timer.xml -->
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC  "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
<bean  id="job1"  class="task1" />
<bean  id="job2"  class="task2" />

 <bean  id="schedule1"  class="org.springframework.scheduling.timer.ScheduledTimerTask" >
        <property name="delay"> <value>5000</value></property>
        <property name="period"><value>2000</value></property>
        <property name="timerTask" >
                <ref bean="job1" />
        </property>
</bean>
<bean  id="schedule2"  class="org.springframework.scheduling.timer.ScheduledTimerTask" >
        <property name="delay"> <value>5000</value></property>
        <property name="period"><value>3000</value></property>
        <property name="timerTask" >
                <ref bean="job2" />
        </property>
</bean>
<bean   id="stt"  class="org.springframework.scheduling.timer.TimerFactoryBean">
        <property name="scheduledTimerTasks">
                <list>
                   <ref  bean="schedule1" />
                   <ref  bean="schedule2" />
                </list>
        </property>
 </bean>
</beans>
//scheduledemo.java
import org.springframework.context.*;
import org.springframework.context.support.*;
import java.io.*;
```

```
public class   scheduledemo
{
      public static void main(String args[]) throws IOException
      {
         ApplicationContext  ctx=new FileSystemXmlApplicationContext("Timer.xml");
      }
}
```

Example: Scheduling using spring classes.

//democlass1.java

```
public class  democlass1
{
   public void  method1()
   {
        System.out.println("welcome to sathya");
    }
}
```

//democlass2.java

```
import java.io.*;
public class  democlass2
{
   public void method2()
   {
        try{
        FileWriter  fp=new FileWriter("demofile.txt",true);
        fp.write(new Date().toString()+"\n");
        fp.close();
        System.out.println("data stored");
        }catch(Exception e){}
    }
}
```

Note: Our classes are not extending any predefined class

Injection Process:

democlass1 -> MethodInvokingTimerTaskFactoryBean -> ScheduledTimerTask -> TimerFactoryBean

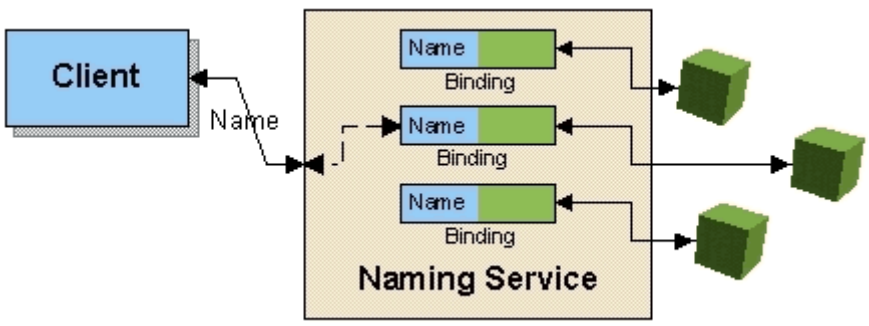
```
<!-- Timer.xml -->
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC  "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
<bean  id="c1"  class="democlass1" />
<bean  id="c2"  class="democlass2" />
<bean id="job1"
class="org.springframework.scheduling.timer.MethodInvokingTimerTaskFactoryBean">
   <property  name="targetObject">
         <ref bean="c1" />
   </property>
   <property  name="targetMethod">
         <value>method1</value>
   </property>
</bean>
<bean id="job2"
class="org.springframework.scheduling.timer.MethodInvokingTimerTaskFactoryBean">
   <property  name="targetObject">
         <ref bean="c2" />
   </property>
   <property  name="targetMethod">
         <value>method2</value>
   </property>
</bean>
 <bean  id="schedule1"  class="org.springframework.scheduling.timer.ScheduledTimerTask" >
         <property name="delay"> <value>5000</value></property>
         <property name="period"><value>2000</value></property>
         <property name="timerTask" >
               <ref bean="job1" />
         </property>
</bean>
<bean  id="schedule2"  class="org.springframework.scheduling.timer.ScheduledTimerTask" >
         <property name="delay"> <value>5000</value></property>
         <property name="period"><value>3000</value></property>
```

```
        <property name="timerTask" >
                <ref bean="job2" />
        </property>
</bean>
<bean   id="stt"  class="org.springframework.scheduling.timer.TimerFactoryBean">
        <property name="scheduledTimerTasks">
                <list>
                   <ref  bean="schedule1" />
                   <ref  bean="schedule2" />
                </list>
        </property>
 </bean>
</beans>
```

Note: Client program is same like earlier example.

JNDI: [Java Naming Directory Interface]

1. When ever we need to give global visibility for any java object then we can use JNDI. So that any program can bidn the object with server from any where through out the world. The same object can be used by some other program through JNDI.
2. JNDI can act as a mediator between server program & client program to exchange objects.
3. To work with JNDI we need to start any application server like weblogic or Jboss.
4. We can do following JNDI operations
    1. bind(id, obj):
        1. We can bind any object with some identification name.
        2. If the object is already binded immediately it throws BindException
    2. rebind(id, obj)
        1. Same as bind() method
        2. But if the object is already exists it replace the exisiting object with new object.
    3. unbind(id): Specific id related object will be immediately removed.
    4. lookup(id): When ever we need to fetch the object through JNDI then we can use it.
    5. list(): returns all the registered objects



Example: Plain JNDI without using Spring

Note:

1. start -> programs -> bea products -> example -> **weblogic server** -> start examples server
2. set the classpath for **weblogic.jar**

//TestJndi.java

```
import javax.naming.*;
import java.util.*;
public class  TestJndi
{
  public static void main(String[] args) throws Exception
   {
          //prepare Jndi properties
             Hashtable ht=new Hashtable();
             ht.put(Context.INITIAL_CONTEXT_FACTORY,
         "weblogic.jndi.WLInitialContextFactory");
          ht.put(Context.PROVIDER_URL,"t3://localhost:7001");
          InitialContext ic=new InitialContext(ht);
         // ic represents connectivity with Naming/Directory registry S/w
         //bind operation
         ic.rebind("today",new Date());
         ic.bind("banana",new String("yellow"));
         //listing
         System.out.println("---->Listing (after binding)");
         NamingEnumeration e=ic.list("");
         while(e.hasMore())
           {
                 NameClassPair np=(NameClassPair)e.next(); // each binding is accessed
                 System.out.println(np.getName()+"-----> "+np.getClassName());
           }
         }//main
}//class
```

```
4126  //TestJndi1.java
4127  import javax.naming.*;
4128  import java.util.*;
4129  public class  TestJndi1
4130  {
4131          public static void main(String[] args) throws Exception
4132          {
4133                  //prepare Jndi properties
4134                  Hashtable ht=new Hashtable();
4135                   ht.put(Context.INITIAL_CONTEXT_FACTORY,
4136                        "weblogic.jndi.WLInitialContextFactory");
4137                   ht.put(Context.PROVIDER_URL,"t3://localhost:7001");
4138                  InitialContext ic=new InitialContext(ht);
4139                  // ic represents connectivity with Naming/Directory registry S/w
4140                  //listing
4141                  System.out.println("Listing");
4142                  NamingEnumeration e=ic.list("");
4143                   while(e.hasMore())
4144                     {
4145                                  NameClassPair np=(NameClassPair)e.next();
4146                                  System.out.println(np.getName()+" ---> "+np.getClassName());
4147                     }
4148              //lookup
4149                  System.out.println("lookup");
4150                  Object obj = ic.lookup("today");
4151                  Date d1=(Date)obj;
4152                  System.out.println("date is "+d1.toString());
4153                  obj = ic.lookup("banana");
4154                  String color=(String)obj;
4155                  System.out.println("Banana color  is "+color);
4156                  //unbind operation
4157                  ic.unbind("banana");
4158                  //listing
4159                  System.out.println("\n\n\nListing---------");
4160                  e=ic.list("");
```

```
4161                    while(e.hasMore())
4162                      {
4163                                  NameClassPair np=(NameClassPair)e.next();
4164                                  System.out.println(np.getName()+" ---> "+np.getClassName());
4165                      }
4166            }//main
4167    }//class
4168    //To execute this application with jboss add  jbossall-client.jar  in the classpath
4169    ht.put(Context.INITIAL_CONTEXT_FACTORY,
4170                          "org.jnp.interfaces.NamingContextFactory");
4171    ht.put(Context.PROVIDER_URL,"jnp://localhost:1099");
```

Note: The rest of steps are same as above program.

Example: Spring JNDI

```
4174    <?xml version="1.0" encoding="UTF-8"?>
4175    <!DOCTYPE beans PUBLIC  "-//SPRING//DTD BEAN//EN"
4176    "http://www.springframework.org/dtd/spring-beans.dtd">
4177    <beans>
4178      <bean id="jt" class="org.springframework.jndi.JndiTemplate">
4179          <constructor-arg>
4180                      <props>
4181                                  <prop key="java.naming.factory.initial">
4182                       weblogic.jndi.WLInitialContextFactory</prop>
4183                                  <prop key="PROVIDER_URL">t3://localhost:7001</prop>
4184             </props>
4185          </constructor-arg>
4186      </bean>
4187      <bean id="test" class="JndiTemplateTest">
4188         <property name="template"><ref bean="jt"/></property>
4189      </bean>
4190    </beans>
4191    //JndiTemplateTest.java
4192    import org.springframework.context.support.*;
4193    import org.springframework.jndi.*;
4194    import java.util.*;
```

```
public class  JndiTemplateTest
{
        static JndiTemplate template;
        public void setTemplate(JndiTemplate template)
        {
                this.template=template;
        }
        public static void main(String[] args) throws Exception
        {
                FileSystemXmlApplicationContext ctx=
                        new FileSystemXmlApplicationContext("DemoCfg.xml");
                System.out.println("bind().....");
                template.bind("today1",new Date());
                template.bind("apple",new String("it is red"));
                System.out.println("lookup operation");
                Date d1=(Date)template.lookup("today1");
                System.out.println("(today1)lookup value is"+d1.toString());
                template.unbind("today1");
                template.rebind("apple",new StringBuffer("it is green"));
                System.out.println("lookup operation");
                StringBuffer s1=(StringBuffer)template.lookup("apple");
                System.out.println("(apple)lookup value is"+s1.toString());
        }
}
```

Note:

1. start -> programs -> bea products -> example -> **weblogic server** -> start examples server
2. set the classpath for weblogic.jar

Example: Plain RMI [Remote Method Invocation]

1. Its a distributed application
2. Server side implementation class will contain logics in the form of methods.
3. Server side business logic methods will be called by client program

```
//ServerInter.java
import java.rmi.*;
public interface ServerInter  extends Remote
{
        public String wish(String uname) throws RemoteException;
}
// ServerImpl.java
import java.rmi.*;
import java.rmi.server.*;
public class ServerImpl   extends UnicastRemoteObject implements  ServerInter
{
        public ServerImpl() throws RemoteException{}
        public String wish(String uname) throws RemoteException
        {
           return "Good Afternoon "+uname;
        }
}
//Server.java
import java.rmi.*;
import java.net.*;
public class  Server
{
  public static void main(String args[]) throws RemoteException,MalformedURLException
  {
        ServerInter i1 = new ServerImpl();
        Naming.rebind("rmi://localhost:1099/first",i1);
        System.out.println("server started..");
  }
}
```

//Client.java

```
import java.rmi.*;
import java.net.*;
public class  Client
{
  public static void main(String args[]) throws
RemoteException,NotBoundException,MalformedURLException
  {
        ServerInter i1=(ServerInter)Naming.lookup("rmi://localhost:1099/first");
        System.out.println(i1.wish("Sai"));
  }
}
```

RmiServer

---

1) In server machine
    1. ServerInter.class
    2. ServerImpl.class
    3. Server.class
2) In client Machine
    1. ServerInter.class
    2. Client.class
    3. ServerImpl_stub.class [We'll get this through rmic (rmi compilation)]

Compilation:

javac *.java

rmic ServerImpl

start rmiregistry [automcatically it starts a rmi registry window. It will be a balck console window. Dont close this. Simply minimise & work. Same jndi operation can be done here. So all the registered objects will be stored in rmi registry]

Execute Server program: java Server

RmiClient

(copy ServerImpl_stub.class & ServerInter.class)

javac Client.java

java Client

Example: Spring RMI

Note: Server side programs & set up every thing is same. But client side programs will be changed as shown below.

//ClientInter.java

```
public interface  ClientInter
{
   public ServerInter  getServerInter();
}
```

// ClientImpl.java

```
public class  ClientImpl implements ClientInter
{
       private  ServerInter si;
       public void setSi(ServerInter si)
       {
                  this.si = si;
       }
      public firstinter  getServerInter()
      {
           return si;
     }
}
```

<!-- first.xml -->

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC  "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
     <bean  id="pfb" class="org.springframework.remoting.rmi.RmiProxyFactoryBean" >
          <property  name="serviceUrl" >
                <value>rmi://localhost:1099/first</value>
          </property>
          <property name="serviceInterface">
                <value>ServerInter</value>
          </property>
     </bean>
```

```
    <bean  id="c1"   class="ClientImpl" >
        <property  name="si">
                <ref  bean="pfb" />
        </property>
    </bean>
</beans>
```

//Client.java

```
import org.springframework.beans.factory.*;
import org.springframework.context.*;
import org.springframework.context.support.*;
import java.rmi.*;
public class  Client
{
  public static void main(String args[]) throws RemoteException
  {
        ApplicationContext ctx=new FileSystemXmlApplicationContext("first.xml");
        BeanFactory factory=(BeanFactory)ctx;
        ClientInter i1=(ClientInter)factory.getBean("c1");
        ServerInter si=i1.getServerInter();
        System.out.println(si.wish("Sai"));
  }
}
```

Example: HttpInvoker

Note:

1. Our business logic classes classes can be placed in classes folder of a web application.
2. Our business logic classes need not to extend any predefined class or need not to implement any predefined interface. They can be perfect POJO's.
3. Same like a common web application. But through concept of HttpInvoker any kind of java applicaiton like console program, desktop application etc.., can give reqest from any where through out the world and can get response.

//httpinter.java

```
public interface httpinter
{
      public String getWeek();
       public String getMonth();
}
```

```
// httpimpl.java
import java.util.*;
public class   httpimpl implements httpinter
{
      public String getWeek()
      {
          String weeks[]=new String[]
{"","Sunday","Monday","Tuesday","Wednesday","Thursday","Friday","Saturday"};
          Calendar  c1=Calendar.getInstance();
          int w=c1.get(Calendar.DAY_OF_WEEK);
          return   "current week :"+weeks[w];
      }
       public String getMonth()
       {
         String months[]=new String[]
{"January","February","March","April","May","June","July","August","September","October","No
vember","December"};
          Calendar  c1=Calendar.getInstance();
           int  m=c1.get(Calendar.MONTH);
           return   "current month :"+months[m];
       }
}
<!-- web.xml -->
<web-app>
  <servlet>
        <servlet-name>remoting</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
   </servlet>
   <servlet-mapping>
        <servlet-name>remoting</servlet-name>
        <url-pattern>/http/*</url-pattern>
   </servlet-mapping>
</web-app>
```

```
<!-- remoting-servlet.xml -->
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC  "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <bean  id="h1"   class="httpimpl" />
  <bean   name="/DemoService"
class="org.springframework.remoting.httpinvoker.HttpInvokerServiceExporter" >
        <property name="service">
            <ref  bean="h1" />
        </property>
        <property  name="serviceInterface" >
            <value>httpinter</value>
        </property>
 </bean>
</beans>
//Client program Setup
//clientinter.java
public interface   clientinter
{
        public httpinter  getInter();
}
//clientimpl .java
public class clientimpl implements   clientinter
{
        private httpinter  httpin;
        public void setHttpin(httpinter f1)
        {
            httpin=f1;
        }
        public httpinter  getInter()
        {
                        return httpin;
        }
}
```

```
<!-- Demo.xml -->
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC  "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
<bean   id="pfb"
class="org.springframework.remoting.httpinvoker.HttpInvokerProxyFactoryBean">
        <property name="serviceUrl">
        <value>http://localhost:2010/httpserver/http/DemoService</value>
        </property>
        <property  name="serviceInterface" >
                <value>httpinter</value>
        </property>
  </bean>
     <bean  id="c1"   class="clientimpl" >
        <property  name="httpin">
                <ref bean="pfb"/>
        </property>
     </bean>
</beans>
//DemoClient.java
import org.springframework.beans.factory.*;
import org.springframework.context.*;
import org.springframework.context.support.*;
public class   DemoClient{
      public static void main(String args[])
      {
        ApplicationContext  ctx=new FileSystemXmlApplicationContext("Demo.xml");
        BeanFactory  factory=(BeanFactory)ctx;
        clientinter c1=(clientinter)factory.getBean("c1");
        httpinter h1=c1.getInter();
        System.out.println(h1.getWeek());
        System.out.println(h1.getMonth());
      }
}
```

Note:

1. Deploy server side ssetup in the form of a web application inside any web server like tomcat or application server like weblogic & start the server & execute the client program.
2. Do not forget to place main jar file & dependency jar file inside "**lib**" folder of a web application.

Example: Working with Spring James Mail Server

Note:

1. The mails what we send & recieve will not be stored in database server like oracle. They will be stored inside mail server.
2. We can have commerical servers & also free servers
3. James mail server is free server. But we can send mails only in same network. We cannot send mails to gmail or yahoomail etc..,
4. Microsoft exchange server is commercial server

Working with mail server:

1. copy james server in C:\james-2.2.0. [in any root directory]

2.start james server by clicking on C:\james-2.2.0\bin\run.bat

3. start -> run -> telnet -> open localhost 4555

login id: root

password: root

type: help -> it gives u a list of commands with explanation.

Note: While you are typing username or password if any mistake ignore it & retype again. It will also recognise back space or space as a part of password. So be careful while typing.

//demointer.java

```
public interface  demointer
{
   public void demo();
}
```

// demoimpl.java

```
import org.springframework.mail.*;
import org.springframework.mail.javamail.*;
public class demoimpl implements   demointer
{
  private  SimpleMailMessage  message;
  private  JavaMailSenderImpl  sender;
```

```
4497   public void setMessage(SimpleMailMessage msg)
4498   {
4499         message=msg;
4500   }
4501   public void setSender(JavaMailSenderImpl sen)
4502   {
4503         sender=sen;
4504   }
4505    public void demo()
4506    {
4507         message.setText("this is first spring mail demo..");
4508          sender.send(message);
4509    }
4510  }
4511  <!--  mail.xml -->
4512  <?xml version="1.0" encoding="UTF-8"?>
4513  <!DOCTYPE beans PUBLIC  "-//SPRING//DTD BEAN//EN"
4514  "http://www.springframework.org/dtd/spring-beans.dtd">
4515  <beans>
4516     <bean  id="msg"  class="org.springframework.mail.SimpleMailMessage" >
4517         <property  name="to"><value>sravanthi</value></property>
4518         <property  name="from"><value>anil</value></property>
4519         <property  name="subject"><value>spring mail1</value></property>
4520    </bean>

4521    <bean  id="s1"  class="org.springframework.mail.javamail.JavaMailSenderImpl">
4522  <!--    <property name="host"><value>m8.hamarashehar.com</value></property>  -->

4523         <property name="host"><value>localhost</value></property>
4524    </bean>

4525     <bean  id="d1"   class="demoimpl" >
4526         <property  name="message">
4527               <ref  bean="msg" />
4528         </property>
```

```
4529            <property  name="sender">
4530                    <ref  bean="s1" />
4531            </property>
4532        </bean>
4533    </beans>
4534    //demoapp.java
4535    import org.springframework.context.*;
4536    import org.springframework.context.support.*;
4537    import org.springframework.beans.factory.*;
4538    public class demoapp
4539    {
4540        public static void main(String args[])
4541        {
4542            ApplicationContext ctx=new FileSystemXmlApplicationContext("mail.xml");
4543            BeanFactory factory=(BeanFactory)ctx;
4544            demointer d1=(demointer)factory.getBean("d1");
4545            d1.demo();
4546            System.out.println("mail sent..");
4547        }
4548    }
4549    //Read mail program
4550    //ReceiveMail.java
4551    // Java Application to receive an E-mail using JAVA Mail API.
4552    import javax.mail.*;
4553    import javax.mail.internet.*;
4554    import java.util.*;
4555    public class ReceiveMail
4556    {
4557       public static void main(String []args) throws Exception
4558        {
4559              Properties p=new Properties();
4560         p.put("mail.transport.protocol","pop");
4561              p.put("mail.pop.host","localhost");
4562              p.put("mail.pop.port","110");
4563              Session session=Session.getInstance(p);
```

```
4564            Store store = session.getStore("pop3");
4565            store.connect("localhost","user_name","pass_word");
4566            Folder myinbox=store.getFolder("INBOX");
4567            myinbox.open(Folder.READ_ONLY);
4568           System.out.println("No.of messages in the inbox:"+myinbox.getMessageCount());
4569            Message message=myinbox.getMessage(1);
4570            message.writeTo(System.out);
4571            myinbox.close(false);
4572            store.close();
4573        }
4574    }
4575    //Example: Spring Mail through gmail composing mails to any gmail id or yahoo mail id
4576    //OrderManager.java
4577    public interface OrderManager {
4578        void placeOrder();
4579    }
4580    //JavaMailOrderManager.java
4581    import javax.mail.Message;
4582    import javax.mail.internet.InternetAddress;
4583    import javax.mail.internet.MimeMessage;
4584    import org.springframework.mail.javamail.JavaMailSender;
4585    import org.springframework.mail.javamail.JavaMailSenderImpl;
4586    public class JavaMailOrderManager implements OrderManager {
4587            private JavaMailSender mailSender;
4588            public void setMailSender(JavaMailSender mailSender) {
4589                    this.mailSender = mailSender;
4590            }
4591            public void placeOrder() {
4592                    try{
4593                            MimeMessage msg = mailSender.createMimeMessage();
4594                            msg.setSubject("Thank you..... :) ");
4595                            msg.setContent("Dear Customer  thank you for placing order. ","text/plain");
4596                            msg.setRecipient(Message.RecipientType.TO, new
4597    InternetAddress("s.varaprasad@yahoo.co.in"));
4598                            this.mailSender.send(msg);
```

```
4599                    }
4600                    catch(Exception ex) {
4601                            ex.printStackTrace();
4602                    }
4603            }
4604    }
4605    <!--  mailsender.xml -->
4606    <?xml version="1.0" encoding="UTF-8"?>
4607    <beans xmlns="http://www.springframework.org/schema/beans"
4608        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4609    xsi:schemaLocation="http://www.springframework.org/schema/beans
4610      http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
4611    <bean id="ms" class="org.springframework.mail.javamail.JavaMailSenderImpl">
4612      <property name="host" value="smtp.gmail.com"/>
4613      <property name="username" value="kanakavaraprasad@gmail.com"/>  <!--Mails will be sent
4614    with same id-->
4615      <property name="password" value="Secret_Pls"/>  <!-- Here we need to give original password
4616    of gmail-->
4617      <property name="javaMailProperties">
4618      <props>
4619              <prop key="mail.transport.protocol">smtp</prop>
4620              <prop key="mail.smtp.auth">true</prop>
4621              <prop key="mail.smtp.port">465</prop>
4622              <prop key="mail.smtp.socketFactory.port">465</prop>
4623              <prop key="mail.smtp.socketFactory.class">javax.net.ssl.SSLSocketFactory</prop>
4624              <prop key="mail.smtp.socketFactory.fallback">false</prop>
4625              <prop key="mail.smtp.quitwait">false</prop>
4626              <prop key="mail.smtp.starttls.enable">true</prop>
4627      </props>
4628      </property>
4629    </bean>
4630    <bean id="orderManager" class="JavaMailOrderManager">
4631      <property name="mailSender" ref="ms"/>
4632    </bean>
4633    </beans>
```

//client.java

```
import org.springframework.core.io.*;
import org.springframework.beans.factory.*;
import org.springframework.beans.factory.xml.*;
class  client
{
        public static void main(String[] args)
        {
                BeanFactory factory=new XmlBeanFactory(new
        FileSystemResource("mailsender.xml"));
                OrderManager om =(OrderManager)factory.getBean("orderManager");
                om.placeOrder();
        }
}
```

Note: Set the classpath for following jar files

1. mail.jar
2. activation.jar

Note: When its needed to send any attachments add the following code:

```
MimeMessage msg = sender.createMimeMessage();
MimeMessageHelper helper = new MimeMessageHelper(msg, true);
FileSystemResource img = new FileSystemResource(new File("alice.gif"));
    helper.addAttachment("alice", img);
    sender.send(msg);
```

Note: In the above code "**sender**" is reference of **"JavaMailSender".**

If wee need to foramt data through html, then we can do it as follows:

```
helper.setText("<html><head></head><body text=green><h1>Hello World!"
            + "</h1></body></html>", true);
```

AOP Module[Aspect Oriented Programming]:

Aspect: An ***Aspect*** is a functionality or a feature that ***cross-cuts [commonly shared] over*** objects.

Eg: example, ***Logging*** and ***Transaction Management*** are the aspects.

```
public void businessOperation(BusinessData data){
   // Logging
   logger.info("Business Method Called");
   // Transaction Management Begin
```

```
    transaction.begin();
    // Do the original business operation here
    transaction.end();
}
```

Join Points defines the various Execution Points where an Aspect can be applied. For example, consider the following piece of code,

```
public void someBusinessOperation(BusinessData data){

    //Method Start -> Possible aspect code here like logging.

    try{
        // Original Business Logic here.
    }catch(Exception exception){
        // Exception -> Aspect code here when some exception is raised.
    }finally{
        // Finally -> Even possible to have aspect code at this point too.
    }

    // Method End -> Aspect code here in the end of a method.
}
```

It is not necessary that an Aspect should be applied to all the possible Join Points.

Pointcut: on which Join Points the Aspects will be applied is called as pointcut.

Advice is the code that implements the **Aspect**

Other Aspect Oriented Programming Languages also provide support for Field Aspect, i.e. intercepting a field before its value gets affected. But Spring provides support only Method Aspect. The following are the different types of aspects available in Spring.

- Before Advice : Executes before business logic method
- After Advice : Executes after business logic method
- Throws Advice : Executes when exception is thrown in a busines logic class
- Around Advice: Executes before & after business logic method.

Around Advice: When ever we need to perform any pre processing logic or post processing logic for our busines logic methods then we can use it by implementing an interace named as MethodInterceptor.

```
4698 //Example: AroundAdvice
4699 //IBusinessLogic.java
4700 public interface IBusinessLogic
4701 {
4702   public void method1();
4703   public void method2();
4704 }
4705 // BusinessLogic.java
4706 public class BusinessLogic implements IBusinessLogic
4707 {
4708     public void method1()
4709     {
4710         System.out.println("execution of method1");
4711     }
4712      public void method2()
4713     {
4714         System.out.println("execution of method2");
4715     }
4716 }
4717 // AroundAdvice.java
4718 import org.aopalliance.intercept.MethodInvocation;
4719 import org.aopalliance.intercept.MethodInterceptor;
4720 public class AroundAdvice implements MethodInterceptor
4721 {
4722   public Object invoke(MethodInvocation i1) throws Throwable
4723   {
4724     System.out.println("Hi.....");
4725     i1.proceed();
4726     System.out.println("Goodbye! ");
4727     return null;
4728   }
4729 }
4730 <!-- springconfig.xml -->
4731 <beans>
```

```
<bean id="targetobj" class="BusinessLogic"/>
  <bean id="adv" class="AroundAdvice"/>

  <bean id="advr" class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
    <property name="advice">
      <ref local="adv"/>
    </property>
    <property name="pattern">
      <value>.*</value>
    </property>
  </bean>
  <bean id="proxyobj" class=
        "org.springframework.aop.framework.ProxyFactoryBean">

         <property name="proxyInterfaces">
      <value>IBusinessLogic</value>
    </property>

    <property name="target">
      <ref local="targetobj"/>
    </property>

    <property name="interceptorNames">
      <list>
        <value>advr</value>
      </list>
    </property>
  </bean>
</beans>
//MainApplication.java
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.FileSystemXmlApplicationContext;
public class MainApplication
{
```

```
4763    public static void main(String [] args)
4764    {
4765     ApplicationContext ctx=
4766                    new FileSystemXmlApplicationContext("springconfig.xml");
4767     IBusinessLogic testObject=(IBusinessLogic) ctx.getBean("proxyobj");
4768     testObject.method1();
4769     testObject.method2();
4770    }
4771  }
4772  //Exmple: Buisness logic method with return value
4773  //IBusinessLogic.java
4774  public interface IBusinessLogic
4775  {
4776    public int method1();
4777  }
4778  // BusinessLogic.java
4779  public class BusinessLogic implements IBusinessLogic
4780  {
4781     public int method1()
4782     {
4783          System.out.println("execution of method1");
4784                  return 1000;
4785     }
4786  }
4787  // AroundAdvice.java
4788  import org.aopalliance.intercept.MethodInvocation;
4789  import org.aopalliance.intercept.MethodInterceptor;
4790  public class AroundAdvice implements MethodInterceptor
4791  {
4792    public Object invoke(MethodInvocation i1) throws Throwable
4793    {
4794      System.out.println("good mng everybody..");
4795      int n=((Integer)i1.proceed()).intValue();
4796      System.out.println("Goodbye! ");
4797                  if(n  <=100)
```

```
                    return  new Integer(0);
                else
                    return  new Integer(n);
                //return  new Integer(2000);
    }
}
```

//Note: What ever the value returned by invoke() method will be returned to client proram. So we can use it when we need to add some value added tax or to give some discount etc..,

```
<!-- springconfig.xml -->
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC  "-//SPRING//DTD BEAN//EN"
   "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
  <bean id="beanTarget" class="BusinessLogic"/>
 <bean id="ad" class="AroundAdvice"/>
  <bean id="advisor" class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
    <property name="advice">
      <ref local="ad"/>
    </property>
    <property name="pattern">
      <value>.*</value>
    </property>
  </bean>
  <bean id="proxyobj" class=
        "org.springframework.aop.framework.ProxyFactoryBean">
    <property name="proxyInterfaces">
      <value>IBusinessLogic</value>
    </property>
    <property name="target">
      <ref local="beanTarget"/>
    </property>
    <property name="interceptorNames">
      <list>
        <value>advisor</value>
```

```
        </list>
      </property>
    </bean>
</beans>
//MainApplication.java
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.FileSystemXmlApplicationContext;
public class MainApplication
{
  public static void main(String [] args)
  {
    ApplicationContext ctx =
                    new FileSystemXmlApplicationContext("springconfig.xml");
    IBusinessLogic testObject = (IBusinessLogic) ctx.getBean("proxyobj");
     int res=testObject.method1();
         System.out.println("test result :"+res);
  }
}
//Example: To do arguments validation before calling business logic method
//IBusinessLogic.java
public interface IBusinessLogic
{
  public void method1(int id,String name);
}
//BusinessLogic.java
public class BusinessLogic implements IBusinessLogic{
    public void method1(int id,String name)
    {
                System.out.println("");
                System.out.println("execution of method1");
                System.out.println(" id :"+id);
                System.out.println("name :"+name);
                System.out.println("");
    }
}
```

```
// AroundAdvice.java
import org.aopalliance.intercept.MethodInvocation;
import org.aopalliance.intercept.MethodInterceptor;

public class AroundAdvice implements MethodInterceptor
{
  public Object invoke(MethodInvocation i1) throws Throwable
  {
        System.out.println("good mng. everybody..");
        int x = ((Integer)i1.getArguments()[0]).intValue();
        String y=(String)i1.getArguments()[1];
        if( x <= 0  )
            i1.getArguments()[0] = new Integer(101);
        if(  y.length() <= 3)
            i1.getArguments()[1]="sathya tech";
        i1.proceed();
        System.out.println("Goodbye! ");
        return null;
  }
}
<!-- springconfig.xml -->
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC  "-//SPRING//DTD BEAN//EN"
 "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
   <bean id="beanTarget" class="BusinessLogic"/>
  <bean id="ad" class="AroundAdvice"/>
  <bean id="advisor" class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
    <property name="advice">
      <ref local="ad"/>
    </property>
    <property name="pattern">
      <value>.*</value>
    </property>
  </bean>
```

```
4901    <bean id="proxyobj" class= "org.springframework.aop.framework.ProxyFactoryBean">
4902      <property name="proxyInterfaces">
4903        <value>IBusinessLogic</value>
4904      </property>
4905      <property name="target">
4906        <ref local="beanTarget"/>
4907      </property>
4908      <property name="interceptorNames">
4909        <list>
4910          <value>advisor</value>
4911        </list>
4912      </property>
4913    </bean>
4914  </beans>
4915  //MainApplication.java
4916  import org.springframework.context.ApplicationContext;
4917  import org.springframework.context.support.FileSystemXmlApplicationContext;
4918  public class MainApplication
4919  {
4920    public static void main(String [] args)
4921    {
4922      ApplicationContext ctx =
4923                    new FileSystemXmlApplicationContext("springconfig.xml");
4924      IBusinessLogic testObject = (IBusinessLogic) ctx.getBean("proxyobj");
4925      testObject.method1(22,"Sai Charan");
4926      testObject.method1(-22,"Kanakadhar");
4927      testObject.method1(-22,"SK");
4928    }
4929  }
4930  //Example: To work with before & anround advice
4931  //IBusinessLogic.java
4932  public interface IBusinessLogic
4933  {
4934    public void method1();
4935  }
```

```
//BusinessLogic.java
public class BusinessLogic implements IBusinessLogic
{
    public void method1()
    {
        System.out.println("");
        System.out.println("execution of method1");
        System.out.println("");
    }
}
// BeforeAdvice1.java
import org.springframework.aop.*;
import java.lang.reflect.*;
public class BeforeAdvice1 implements MethodBeforeAdvice
{
        public void before(Method method, Object[] args, Object target)
throws Throwable
        {
                System.out.println("before advice");
        }
}
// AroundAdvice2.java
import org.aopalliance.intercept.MethodInvocation;
import org.aopalliance.intercept.MethodInterceptor;
public class AroundAdvice2 implements MethodInterceptor
{
  public Object invoke(MethodInvocation i1) throws Throwable
  {
    System.out.println(" AroundAdvice : good mng everybody..");
    i1.proceed();
    System.out.println("AroundAdvice : Goodbye! ");
    return null;
  }
}
```

```
<!-- springconfig.xml -->
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC  "-//SPRING//DTD BEAN//EN"
  "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
   <bean id="adv1" class="BeforeAdvice1"/>
   <bean id="adv2" class="AroundAdvice2"/>
   <bean id="targetobj" class="BusinessLogic"/>
   <bean id="advr1"
class="org.springframework.aop.support.NameMatchMethodPointcutAdvisor">
      <property name="advice">
        <ref local="adv1"/>
      </property>
      <property name="mappedName">
        <value>method1</value>
      </property>
   </bean>
 <bean id="advr2" class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
      <property name="advice">
        <ref local="adv2"/>
      </property>
      <property name="pattern">
        <value>.*</value>
      </property>
   </bean>
   <bean id="proxyobj" class= "org.springframework.aop.framework.ProxyFactoryBean">
      <property name="proxyInterfaces">
        <value>IBusinessLogic</value>
      </property>
      <property name="target">
        <ref local="targetobj"/>
      </property>
      <property name="interceptorNames">
        <list>
          <value>advr2</value>
```

```
            <value>advr1</value>
          </list>
        </property>
    </bean>
</beans>
```

//MainApplication.java

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.FileSystemXmlApplicationContext;
public class MainApplication
{
  public static void main(String [] args)
  {
    ApplicationContext ctx =
                    new FileSystemXmlApplicationContext("springconfig.xml");
    IBusinessLogic testObject = (IBusinessLogic) ctx.getBean("proxyobj");
    testObject.method1();
  }
}
```

Example: Working with all 4 advices [Before, After Around & Throws]

Note: Here we will work with log4j [logging 4 java]

Log4j: is a logging framework which is used to identify the control flow through log files. We can know where the problem is arised. Normally we can display messages on server console using s.o.p() statements. But those statements will be skipped soon & we cannot capture them. We can capture complete logs inforamtion through this log4j framework & we can store in the form of files. So we ever we need to we can open log files & can refer.

Note: To work with this we need to set the classpath for **log4j.jar**

// Adder.java

```
package aop;
public interface Adder {
   public int add(int a,int b);
}
```

//AdderImpl .java

```
package aop;
public class AdderImpl implements Adder {
   public int add(int a, int b){
      return a+b;
```

```
5041    }
5042  }
5043  //LogBeforeCallAdvice.java
5044  package aop;
5045  import java.lang.reflect.Method;
5046  import org.springframework.aop.MethodBeforeAdvice;
5047  import org.apache.log4j.*;
5048  public class LogBeforeCallAdvice implements MethodBeforeAdvice{
5049      Logger logger = Logger.getLogger(LogBeforeCallAdvice.class);
5050          public    LogBeforeCallAdvice()
5051           {
5052                  SimpleLayout layout = new SimpleLayout();
5053                  FileAppender appender = null;
5054                  try
5055                  {
5056                    appender = new FileAppender(layout,"output1.txt",true);
5057                  }
5058                 catch(Exception e) {}
5059                  logger.addAppender(appender);
5060                  logger.setLevel((Level) Level.DEBUG);
5061    }
5062    public void before(Method method, Object[] args, Object target)throws Throwable {
5063     logger.info("Before Calling the Method at: "+new java.util.Date());
5064    }
5065  }
5066  // LogAfterReturningAdvice.java
5067  package aop;
5068  import java.lang.reflect.Method;
5069  import org.springframework.aop.AfterReturningAdvice;
5070  import org.apache.log4j.*;
5071  public class LogAfterReturningAdvice implements AfterReturningAdvice{
5072      Logger logger = Logger.getLogger(LogAfterReturningAdvice.class);
5073          public LogAfterReturningAdvice()
5074           {
5075                  SimpleLayout layout = new SimpleLayout();
```

```
                FileAppender appender = null;
                try
                {
                   appender = new FileAppender(layout,"output1.txt",true);
                }
                catch(Exception e) {}
                logger.addAppender(appender);
                logger.setLevel((Level) Level.DEBUG);
    }
    public void afterReturning(Object returnValue, Method method, Object[] args, Object target)
throws Throwable {
        logger.info("After Normal Return from Method at "
                                                +new java.util.Date());
                }
}
// LogAroundAdvice.java
package aop;
import org.aopalliance.intercept.*;
public class LogAroundAdvice implements MethodInterceptor{

    public Object invoke(MethodInvocation i1) throws Throwable {
        Object arguments[] = i1.getArguments();
        int number1 = ((Integer)arguments[0]).intValue();
        int number2 = ((Integer)arguments[1]).intValue();
        if (number1 == 0 && number2 == 0){
            throw new Exception("Dont know how to add 0 and 0!!!");
        }
        return i1.proceed();
    }
}
// LogAfterThrowsAdvice.java
package aop;
import java.lang.reflect.Method;
import org.springframework.aop.ThrowsAdvice;
public class LogAfterThrowsAdvice implements ThrowsAdvice{
```

```
5110    public void afterThrowing(Method method, Object[] args, Object target,
5111    Exception exception){
5112     System.out.println("Exception is thrown on method "+method.getName());
5113     System.out.println("Exception that is  raised is  " +exception.toString());
5114    }
5115 }
5116 <!-- aop-test.xml -->
5117 <?xml version="1.0" encoding="UTF-8"?>
5118 <!DOCTYPE beans PUBLIC  "-//SPRING//DTD BEAN//EN"
5119     "http://www.springframework.org/dtd/spring-beans.dtd">
5120 <beans>
5121    <!-- Advices -->
5122    <bean id = "beforeCall" class = "aop.LogBeforeCallAdvice" />
5123    <bean id = "afterCall" class = "aop.LogAfterReturningAdvice" />
5124    <bean id = "throwCall" class = "aop.LogAfterThrowsAdvice" />
5125    <bean id = "aroundCall" class = "aop.LogAroundAdvice" />
5126    <!-- Implementation Class -->
5127    <bean id = "adderImpl" class = "aop.AdderImpl" />
5128    <!-- Proxy Implementation Class -->
5129    <bean id="proxy" class=
5130                "org.springframework.aop.framework.ProxyFactoryBean">
5131       <property name = "proxyInterfaces">
5132          <value>aop.Adder</value>
5133       </property>
5134       <property name = "interceptorNames">
5135          <list>
5136             <value>beforeCall</value>
5137             <value>afterCall</value>
5138             <value>throwCall</value>
5139             <value>aroundCall</value>
5140          </list>
5141       </property>
5142       <property name = "target">
5143          <ref bean = "adderImpl"/>
5144       </property>
```

```
    </bean>
</beans>
```

//AopTestClient .java

```
package aop;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.*;
public class AopTestClient {
    public static void main(String args[]){
        Resource resource = new FileSystemResource("aop-test.xml");
        BeanFactory factory = new XmlBeanFactory(resource);
        Adder adder = (Adder)factory.getBean("proxy");
        int result = adder.add(0, 0);
//        int result = adder.add(10, 20);
        System.out.println("Result = " + result);
    }
}
```

//Example: Integration spring – servlet & jdbc

Note:*jar files required in WEB-INF\lib folder & classpath are spring.jar, commons-logging.jar, classes12.jar*

<!-- index.html -->

```
<frameset rows="30%,*" border = "0">
  <frame src="Search.jsp">
  <frame name="resultframe">
</frameset>
```

<!-- Search.jsp -->

```
<body bgcolor="pink" >
  <form action="controller" target="resultframe">
   <b>Select job</b>
            <select name="job">
       <option>CLERK</option>
       <option>ANALYST</option>
       <option>SALESMAN</option>
       <option>MANAGER</option>
      </select>
```

```
5180    <input type="submit" value="search">
5181   </form>
5182 </body>
5183 <!-- Result.jsp -->
5184 <%@page import="java.util.*, p1.EmpBean"%>
5185 <table>
5186 <% ArrayList list=(ArrayList)request.getAttribute("result");
5187   if(list!=null)
5188   { %>
5189    <%System.out.println("No. of records are : "+list.size()); %>
5190    <tr>
5191      <th>ID</th>
5192      <th>Name</th>
5193      <th>Desg</th>
5194      <th>Salary</th>
5195    </tr>
5196    <%for(int i=0;i<list.size();++i)
5197      {
5198      EmpBean eb=(EmpBean)list.get(i); %>
5199      <tr>
5200         <td><%=eb.getId()%></td>
5201         <td><%=eb.getName()%></td>
5202         <td><%=eb.getDesg()%></td>
5203         <td><%=eb.getBsal()%></td>
5204      </tr>
5205      <% } %>
5206      </table>
5207     <% } %>
5208 <!-- web.xml -->
5209 <?xml version="1.0" encoding="UTF-8"?>
5210 <web-app>
5211  <servlet>
5212    <description>This is the description of my J2EE component</description>
5213    <display-name>This is the display name of my J2EE component</display-name>
5214    <servlet-name>MainServlet</servlet-name>
```

```
    <servlet-class>MainServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>MainServlet</servlet-name>
    <url-pattern>/controller</url-pattern>
  </servlet-mapping>
</web-app>
```

//EmpBean.java

```
package p1;
public class EmpBean {
	int id;
	String name,desg;
	float bsal;
	public EmpBean()
	{
		System.out.println("EmpBean Constructor");
	}
	public float getBsal() {
		return bsal;
	}
	public void setBsal(float bsal) {
		this.bsal = bsal;
	}
	public String getDesg() {
		return desg;
	}
	public void setDesg(String desg) {
		this.desg = desg;
	}
	public int getId() {
		return id;
	}
	public void setId(int id) {
		this.id = id;
	}
```

```
5250        public String getName() {
5251                return name;
5252        }
5253        public void setName(String name) {
5254                this.name = name;
5255        }
5256  }
5257  //Model .java
5258  import java.util.*;
5259  public interface Model {
5260        public ArrayList search(String desg);
5261  }
5262  // ModelBean.java
5263  import java.util.*;
5264  import javax.sql.*;
5265  import java.sql.*;
5266  import p1.EmpBean;
5267  public class ModelBean implements Model{
5268        public DataSource ds = null;
5269        public void setDs(DataSource ds) {
5270                this.ds = ds;
5271        }
5272        public ArrayList search(String desg)
5273        {
5274                System.out.println("search() of ModelBean class");
5275                ArrayList al=new ArrayList();
5276                try {
5277                        Connection con=ds.getConnection();
5278                        String qry = "select empno, ename, job, sal from emp where job=?";
5279                        PreparedStatement ps=con.prepareStatement(qry);
5280                        ps.setString(1,desg);
5281                        ResultSet rs=ps.executeQuery();
5282                        while(rs.next())
5283                        {
5284                                EmpBean eb=new EmpBean();
```

```
                                eb.setId(rs.getInt(1));
                                eb.setName(rs.getString(2));
                                eb.setDesg(rs.getString(3));
                                eb.setBsal(rs.getFloat(4));
                                al.add(eb);
                        }
                } catch (SQLException e) {
                        e.printStackTrace();
                }
                return al;
        }
}
<!-- modelconfig.xml -->
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
        "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
  <bean id="ds" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
            <property name="driverClassName">
                    <value>oracle.jdbc.driver.OracleDriver</value>
            </property>
            <property name="url">
              <value>jdbc:oracle:thin:@localhost:1521:sathya</value>
            </property>
            <property name="username"><value>scott</value></property>
            <property name="password"><value>tiger</value></property>
  </bean>
          <bean id="mdb" class="ModelBean" autowire="byName"/>
</beans>
// MainServlet.java
import java.io.IOException;
import java.util.ArrayList;
import javax.servlet.*;
import javax.servlet.http.*;
import org.springframework.beans.factory.xml.*;
```

```
import org.springframework.core.io.*;
public class MainServlet extends HttpServlet {
	Model mod1=null;
	public void init()
	{
		System.out.println("Init Method");
		ClassPathResource res=new ClassPathResource("modelconfig.xml");
		XmlBeanFactory factory=new XmlBeanFactory(res);
		mod1=(Model)factory.getBean("mdb");
	}//init
	public void destroy() {
		System.out.println("destroy()");
		mod1=null;
	}
	public void doGet(HttpServletRequest request, HttpServletResponse response)
			throws ServletException, IOException {
		System.out.println("doGet() of MainServlet");
		try {
			String job=request.getParameter("job").trim();
			ArrayList al=mod1.search(job);
			request.setAttribute("result",al);
			RequestDispatcher rd=request.getRequestDispatcher("Result.jsp");
			if(rd!=null)
			  rd.forward(request,response);
		} catch (ServletException e) {
			e.printStackTrace();
		} catch (IOException e) {
			e.printStackTrace();
		}
	}
	public void doPost(HttpServletRequest request, HttpServletResponse response)
			throws ServletException, IOException {
		System.out.println("doPost() of MainServlet");
			try {
				doGet(request,response);
```

```
                        } catch (ServletException e) {
                                e.printStackTrace();
                        } catch (IOException e) {
                                e.printStackTrace();
                        }
        }//doPost
}//MainServlet
//Example: Spring – Hibernate -Servlet
<!-- web.xml -->
<web-app>
  <servlet>
        <servlet-name>select</servlet-name>
        <servlet-class>selectservlet</servlet-class>
   </servlet>
   <servlet-mapping>
        <servlet-name>select</servlet-name>
        <url-pattern>/selectaction</url-pattern>
   </servlet-mapping>
</web-app>
//demointer.java
import org.hibernate.*;
public interface   demointer
{
      public SessionFactory   getFactory();
}
// demoimpl.java
import org.hibernate.*;
public class demoimpl implements demointer
{     private SessionFactory  sesfact;
      public void setSesfact(SessionFactory f1)
      {
           sesfact=f1;
      }
```

```
5388        public SessionFactory   getFactory()
5389        {
5390           return sesfact;
5391         }
5392  }
```

5393 **Note:** Create following table in oracle & insert some reocrds So you can see stored records as a
5394 report

```
5395  create table users
5396  (
5397           userid number(5) primary key,
5398           uname varchar2(20),
5399           role varchar2(20)
5400  );
5401  //User.java
5402  public class User
5403  {
5404           private int uid;
5405           private String uname,role;
5406           public void setUid(int n)
5407           {
5408                    uid=n;
5409           }
5410           public int getUid(){ return uid; }
5411           public void setUname(String s){ uname=s; }
5412           public String getUname(){ return uname; }
5413           public void setRole(String r){ role=r; }
5414           public String getRole(){  return role; }
5415  }
5416  <!-- User.hbm.xml -->
5417  <?xml version="1.0"?>
5418  <!DOCTYPE hibernate-mapping PUBLIC
5419  "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
5420  "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
5421  <hibernate-mapping>
```

```
5422      <class    name="User"   table="users" >
5423          <id   name="uid"  column="userid"  />
5424          <property   name="uname" />
5425          <property   name="role" />
5426      </class>
5427  </hibernate-mapping>
5428  <!-- ApplicationContext.xml -->
5429  <?xml version="1.0" encoding="UTF-8"?>
5430  <!DOCTYPE beans PUBLIC  "-//SPRING//DTD BEAN//EN"
5431  "http://www.springframework.org/dtd/spring-beans.dtd">
5432  <beans>
5433    <bean id="myDataSource"
5434       class="org.springframework.jdbc.datasource.DriverManagerDataSource">
5435     <property name="driverClassName">
5436       <value>oracle.jdbc.driver.OracleDriver</value>
5437     </property>
5438     <property name="url">
5439       <value>jdbc:oracle:thin:@localhost:1521:sathya</value>
5440     </property>
5441     <property name="username">
5442       <value>scott</value>
5443     </property>
5444     <property name="password">
5445       <value>tiger</value>
5446     </property>
5447    </bean>
5448  <bean id="mySessionFactory"
5449  class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
5450  <property name="dataSource" ref="myDataSource"/>
5451  <property name="mappingResources">
5452          <list>
5453                <value>User.hbm.xml</value>
5454          </list>
5455  </property>
```

```
<property name="hibernateProperties">
   <props>
        <prop key="hibernate.dialect">org.hibernate.dialect.Oracle9Dialect</prop>
    </props>
</property>
</bean>
 <bean   id="d1"  class="demoimpl">
   <property name="sesfact">
      <ref local="mySessionFactory"/>
   </property>
 </bean>
</beans>
```

//selectservlet.java

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import org.hibernate.*;
import java.util.*;
import org.springframework.beans.factory.*;
import org.springframework.context.*;
import org.springframework.context.support.*;
public class  selectservlet   extends HttpServlet
{
 public void service(HttpServletRequest request,HttpServletResponse response)
  throws ServletException,IOException
  {       PrintWriter out=response.getWriter();

          try{
            ApplicationContext ctx=new
ClassPathXmlApplicationContext("ApplicationContext.xml");
             BeanFactory  factory=(BeanFactory)ctx;
             demointer d1=(demointer)factory.getBean("d1");
            SessionFactory sf=d1.getFactory();
              Session ses=sf.openSession();
               Query query=ses.createQuery("from  User");
```

```
5490            Iterator i1=query.iterate();
5491            out.println("<body bgcolor=#ffffcc text=red>");
5492           out.println("<h1><center>all users</h1><hr><br><h3>");
5493          out.println("<table width=80% border=2>");
5494           while(i1.hasNext())
5495           {
5496                  User u1=(User)i1.next();
5497                  out.println("<tr><td>"+u1.getUid()+" <td>"+u1.getUname()+" <td>"+u1.getRole()
5498 +"</tr>");
5499           }
5500           out.println("</table>");
5501                  ses.close();
5502          }catch(HibernateException e)
5503           {
5504                  out.println(e);
5505                  e.printStackTrace();
5506           }
5507    }
5508 }
```

5509 **Note:** Set the classpath for:

5510 1. 8 hibernate jars [1 main har + 7 dependncy jars]
5511 2. 2 spring jars [1 main + 1 dependency jar]
5512 3. oracle database specific jar

5513 **Note:** After deplying & starting web or application server give url as follows in browser:

5514 http://localhost:2010/SpringHibernateServlet/selectaction

5515 **Spring MVC:**

5516 **Example: Spring MVC example with SimpleFormController class**

5517 **<!-- index.jsp -->**

```
5518 <jsp:forward   page="/newpro.htm"  />
5519 <!-- web.xml -->
5520 <web-app>
5521     <servlet>
5522       <servlet-name>productaction</servlet-name>
5523       <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
```

```
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
       <servlet-name>productaction</servlet-name>
       <url-pattern>*.htm</url-pattern>
    </servlet-mapping>
    <taglib>
        <taglib-uri>spring-tld</taglib-uri>
        <taglib-location>/WEB-INF/spring.tld</taglib-location>
     </taglib>
</web-app>
```

Note: we need to name spring configuration file name as follows:

<front controller class servlet name>-servlet.xml

Eg: In above example our front controller class "**DispatcherServlet"** is named as "**productaction"** so we have named the spring configuration file as "**productaction-servlet.xml".**

```
<!-- productaction-servlet.xml -->
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC  "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <bean  id="simpleUrlMappings"
        class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping" >
          <property  name="mappings">
                <props>
                      <prop  key="/newpro.htm">newproController</prop>
                </props>
           </property>
    </bean>
    <bean  id="newproController"   class="NewProductController" >
                <property name="sessionForm" > <value>true</value></property>
                <property name="commandName" > <value>product</value></property>
                <property name="commandClass" > <value>Product</value></property>
                <property name="formView" > <value>newproduct</value></property>
                <property name="successView" > <value>showproduct</value></property>
    </bean>
```

```
        <bean id="viewResolver" class =
           "org.springframework.web.servlet.view.InternalResourceViewResolver">
           <property name="viewClass">
                   <value>org.springframework.web.servlet.view.JstlView</value>
           </property>
           <property name ="prefix"><value>/</value></property>
           <property name ="suffix"><value>.jsp</value></property>
        </bean>
</beans>
<!-- newproduct.jsp -->
<%@ page language="java" %>
<%@ taglib  prefix="spring"  uri="spring-tld"  %>
<html>
<body bgcolor=#ffffcc text=green>
<h1><center>new product information</h1><hr><br><h3>
<form method="post" >
enter product id :
<spring.bind  path="product.pid" >
<input type=text name=pid />
</spring.bind>
<br><br>
enter product name :
<spring.bind  path="product.pname" >
<input type=text name=pname />
</spring.bind>
<br><br>
enter product price :
<spring.bind  path="product.price" >
<input type=text name=price />
</spring.bind>
<br><br>
<input type=submit value=send />
</form>
</body>
</html>
```

```
//Product.java
public class Product
{
        private  int pid;
         private String pname;
         private double price;

         public void setPid(int n){ pid=n; }
         public int getPid(){ return pid; }
         public void setPname(String s){ pname=s; }
         public String getPname(){ return pname; }
         public void setPrice(double p){ price=p; }
         public double getPrice(){ return price; }
}
//NewProductController.java
import org.springframework.web.servlet.mvc.*;
import org.springframework.web.servlet.*;
import   org.springframework.validation.*;
import javax.servlet.http.*;
public class   NewProductController  extends SimpleFormController
{
    public ModelAndView  onSubmit(HttpServletRequest request,HttpServletResponse
response,Object command,BindException e) throws Exception
    {
          Product p=(Product)command;
         String str="Product id :"+p.getPid()+" <br>  "
                  +"product name :"+p.getPname()+"<br>  "
                 +"Price :"+p.getPrice();
                 System.out.println(str);
           request.setAttribute("productinfo",str);
           return new ModelAndView(getSuccessView());
         //   return new ModelAndView("showproduct");
    }
}
```

```
<!-- showproduct.jsp -->
<%@ page language="java" %>
<html>
<body bgcolor=#ffffcc text=green>
<h1><center>new product information</h1><hr><br><h3>
 <%=  request.getAttribute("productinfo") %>
</body>
</html>
```

Example: Spring MVC example with "Controller" interface

```
<!-- index.jsp -->
<jsp:forward   page="/display.form"  />
<!-- web.xml -->
<web-app>
  <servlet>
   <servlet-name>accaction</servlet-name>
   <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
   <load-on-startup>1</load-on-startup>
   </servlet>
 <servlet-mapping>
 <servlet-name>accaction</servlet-name>
 <url-pattern>*.form</url-pattern>
 </servlet-mapping>
 <taglib>
    <taglib-uri>spring-tld</taglib-uri>
    <taglib-location>/WEB-INF/spring.tld</taglib-location>
  </taglib>
</web-app>
<!-- accaction-servlet.xml -->
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC  "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
     <bean  id="simpleUrlMappings"
          class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping" >
            <property  name="mappings">
```

```
5662                    <props>
5663                        <prop  key="/display.form">displayaccController</prop>
5664                        <prop  key="/newacc.form">newaccController</prop>
5665                     </props>
5666              </property>
5667       </bean>
5668       <bean  id="displayaccController"   class="DisplayAccountsController" >
5669            <property  name="dsource">
5670                    <ref bean="ds" />
5671            </property>
5672    </bean>
5673     <bean  id="newaccController"   class="NewAccountController" >
5674            <property name="sessionForm" > <value>true</value></property>
5675            <property name="commandName" > <value>account</value></property>
5676            <property name="commandClass" > <value>Account</value></property>
5677            <property name="formView" > <value>newaccount</value></property>
5678            <property name="successView" > <value>index</value></property>
5679            <property  name="dsource">
5680                        <ref bean="ds" />
5681            </property>
5682       </bean>
5683        <bean id="viewResolver" class =
5684            "org.springframework.web.servlet.view.InternalResourceViewResolver">
5685            <property   name="viewClass">
5686                    <value>org.springframework.web.servlet.view.JstlView</value>
5687            </property>
5688            <property name ="prefix"><value>/</value></property>
5689            <property name ="suffix"><value>.jsp</value></property>
5690         </bean>
5691    <bean   id="ds" class="org.springframework.jdbc.datasource.DriverManagerDataSource" >
5692            <property  name="driverClassName">
5693                    <value>oracle.jdbc.driver.OracleDriver</value></property>
5694            <property  name="url">
5695                    <value>jdbc:oracle:thin:@localhost:1521:sathya</value>
5696            </property>
```

```
        <property  name="username" ><value>scott</value></property>
        <property  name="password" ><value>tiger</value></property>
    </bean>
</beans>
//Account.java
public class Account
{
        private  int acno;
         private String acname;
         private double balance;
         public Account(){}
         public Account(int a,String b,double c)
         {
                 acno=a;
                 acname=b;
                 balance=c;
         }
         public void setAcno(int n){ acno=n; }
         public int getAcno(){ return acno; }
         public void setAcname(String s){ acname=s; }
         public String getAcname(){ return acname; }
         public void setBalance(double p){ balance=p; }
         public double getBalance(){ return balance; }
}
// DisplayAccountsController.java
import org.springframework.web.servlet.mvc.*;
import org.springframework.web.servlet.*;
import   org.springframework.validation.*;
import javax.servlet.http.*;
import javax.sql.*;
import java.util.*;
import org.springframework.jdbc.core.*;
import org.springframework.jdbc.support.rowset.*;
```

```
public class   DisplayAccountsController implements Controller
{
   private   DataSource dsource;
    public void setDsource(DataSource ds)
    {
         dsource=ds;
    }
   public ModelAndView  handleRequest(HttpServletRequest request,HttpServletResponse
response)
   {
                List  accs=new ArrayList();
         try{
           JdbcTemplate  jt=new JdbcTemplate(dsource);
           SqlRowSet rs=jt.queryForRowSet("select * from bank");
                while(rs.next())
                {
                        accs.add(new Account(rs.getInt(1),rs.getString(2),rs.getDouble(3)));
                }
         }catch(Exception e){ System.out.println(e); }
         return new ModelAndView("displayaccounts","listaccounts",accs);
    }
}
<!-- displayaccounts.jsp -->
<%@ page language="java" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core" %>
<html>
<body bgcolor=#ffffcc text=green>
<h1><center>all accounts information</h1><hr><br><h3>
<table width=80% border=2>
<tr><th>a/c number<th>a/c holder name<th> balance</tr>
<c:forEach  items="${listaccounts}"  var="acc" >
  <tr>
         <td><c:out   value="${acc.acno}" />
         <td><c:out   value="${acc.acname}" />
         <td><c:out   value="${acc.balance}" />
```

```
5765    </tr>
5766  </c:forEach>
5767  </table>
5768  <center><br><br>
5769  <a href="/AccountsMvc/newacc.form">new account</a>
5770  </body>
5771  </html>
5772  <!-- newaccount.jsp -->
5773  <%@ page language="java" %>
5774  <%@ taglib  prefix="spring"  uri="spring-tld"  %>
5775  <html>
5776  <body bgcolor=#ffffcc text=green>
5777  <h1><center>new account information</h1><hr><br><h3>
5778  <form method="post" >
5779  enter a/c number :
5780  <spring.bind  path="account.acno" >
5781  <input type=text name=acno />
5782  </spring.bind>
5783  <br><br>
5784  enter a/c holder name :
5785  <spring.bind  path="account.acname" >
5786  <input type=text name=acname />
5787  </spring.bind>
5788  <br><br>
5789  enter open balance :
5790  <spring.bind  path="account.balance" >
5791  <input type=text name=balance />
5792  </spring.bind>
5793  <br><br>
5794  <input type=submit value=save />
5795  </form>
5796  </body>
5797  </html>
```

```
// NewAccountController.java
import org.springframework.web.servlet.mvc.*;
import org.springframework.web.servlet.*;
import   org.springframework.validation.*;
import javax.servlet.http.*;
import javax.sql.*;
import org.springframework.jdbc.core.*;

public class   NewAccountController  extends SimpleFormController
{
    private   DataSource dsource;
    public void setDsource(DataSource ds)
    {
                        dsource=ds;
    }
     public ModelAndView  onSubmit(HttpServletRequest request,HttpServletResponse
response,Object command,BindException e1) throws Exception
     {
          Account a1=(Account)command;
           try{
                JdbcTemplate jt=new JdbcTemplate(dsource);
                jt.update("insert into bank values(?,?,?)",new Object[]{new
          Integer(a1.getAcno()),a1.getAcname(),new Double(a1.getBalance())});
           }catch(Exception e){ System.out.println(e); }
             return new ModelAndView(getSuccessView());
     }
}
```

Example: Pure Struts Application

```
<!-- web.xml -->
<web-app>
  <!-- Action Servlet Configuration -->
  <servlet>
    <servlet-name>action</servlet-name>
    <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
    <init-param>
      <param-name>config</param-name>
```

```
5833      <param-value>/WEB-INF/struts-config.xml</param-value>
5834    </init-param>
5835    <load-on-startup>1</load-on-startup>
5836  </servlet>
5837  <!-- Action Servlet Mapping -->
5838  <servlet-mapping>
5839    <servlet-name>action</servlet-name>
5840    <url-pattern>*.do</url-pattern>
5841  </servlet-mapping>
5842  <!-- The Usual Welcome File List -->
5843  <welcome-file-list>
5844    <welcome-file>register.jsp</welcome-file>
5845  </welcome-file-list>
5846  <!-- Struts Tag Library Descriptors -->
5847  <taglib>
5848    <taglib-uri>/hi-friends-geve</taglib-uri>
5849    <taglib-location>/WEB-INF/struts-html.tld</taglib-location>
5850  </taglib>
5851 </web-app>
5852 <!-- register.jsp -->
5853 <%@ taglib uri="/hi-friends-gmg" prefix="html" %>
5854 <html:html>
5855         <html:form action = "/register">
5856                 <table border = 0 align = center>
5857                         <tr>
5858                                 <th>UserName
5859                                 <td><html:text property = "username"/>
5860                         </tr>
5861                         <tr>
5862                                 <th>PassWord
5863                                 <td><html:password property = "password"/>
5864                         </tr>
5865                         <tr>
5866                                 <td colspan = 2 align = center>
5867                                         <html:submit value = "Register"/>
```

```
                    </tr>
                </table>
        </html:form>
</html:html>
<!-- struts-config.xml -->
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE struts-config PUBLIC "-//Apache Software Foundation//DTD Struts Configuration
1.1//EN"                    "http://jakarta.apache.org/struts/dtds/struts-config_1_1.dtd">
<struts-config>
        <form-beans>
                <form-bean name="registerForm" type="app.RegisterForm"/>
        </form-beans>
        <action-mappings>
          <action name="registerForm" path="/register" type="app.RegisterAction">
                <forward name="ok" path="/success.jsp"/>
                <forward name="fail" path="/failure.jsp"/>
          </action>
        </action-mappings>
</struts-config>
//RegisterForm.java
package app;
import org.apache.struts.action.*;
public class RegisterForm extends ActionForm
{
        private String username = null;
        private String password = null;
        public void setUsername(String username)
        {
                this.username = username;
        }
        public String getUsername()
        {
                return username;
        }
```

```
5902        public void setPassword(String password)
5903        {
5904                this.password = password;
5905        }
5906        public String getPassword()
5907        {
5908                return password;
5909        }
5910  }
5911  // RegisterAction.java
5912  package app;
5913  import org.apache.struts.action.*;
5914  import javax.servlet.http.*;
5915  public class RegisterAction extends Action
5916  {
5917        public ActionForward execute(ActionMapping mapping,ActionForm
5918  form,HttpServletRequest request,HttpServletResponse response)throws Exception
5919        {
5920                RegisterForm rf = (RegisterForm)form;
5921                String user = rf.getUsername();
5922                String pass = rf.getPassword();
5923                if(user.equals("sathya") && pass.equals("java"))
5924                        return mapping.findForward("ok");
5925                else
5926                        return mapping.findForward("fail");
5927        }
5928  }
5929  <!-- success.jsp -->
5930  <html>
5931        <body>
5932                <center>
5933                        <font size = 5 color = green>Login Successfull</font>
5934                </center>
5935        </body>
5936  </html>
```

```
<!-- failure.jsp -->
<html>
        <body>
                <center>
                        <font size = 5 color = red>Login Failure</font><br>
                        <a href = "register.jsp">Try Again</a>
                </center>
        </body>
</html>
```

Note: Place struts related jar files inside "lib" folder of WEB-INF.

```
//Example: Integeration of struts & Spring
<!-- web.xml -->
<web-app>
  <servlet>
        <servlet-name>action</servlet-name>
        <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
        <init-param>
                <param-name>config</param-name>
                <param-value>/WEB-INF/struts-config.xml</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
      </servlet>
      <servlet-mapping>
        <servlet-name>action</servlet-name>
        <url-pattern> *.do </url-pattern>
      </servlet-mapping>
     <welcome-file-list>
        <welcome-file>index.jsp</welcome-file>
     </welcome-file-list>
     <taglib>
        <taglib-uri>struts-html</taglib-uri>
        <taglib-location>/WEB-INF/struts-html.tld</taglib-location>
     </taglib>
</web-app>
```

```
<!-- index.jsp -->
<%@  page language="java" %>
<%@ taglib  prefix="html"  uri="struts-html"  %>
<html>
<body bgcolor=#ffffcc text=green>
<h1><center>struts framewrok</h1><hr><br><h3>
<html:form action="/demo">
  enter user name :<html:text  property="uname"  /> <br><br>
  <html:submit  value="wish" />
</html:form>
</body>
</html>
<!-- struts-config.xml -->
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE struts-config PUBLIC "-//Apache Software Foundation//DTD Struts Configuration
1.1//EN"
                        "http://jakarta.apache.org/struts/dtds/struts-config_1_1.dtd">
<struts-config>
    <form-beans>
        <form-bean   name="demofrm"    type="demopack.DemoForm" />
     </form-beans>
    <action-mappings>
        <action  path="/demo"  name="demofrm"
        type="org.springframework.web.struts.DelegatingActionProxy" >
            <forward  name="success"  path="/result.jsp" />
        </action>
    </action-mappings>
        <plug-in  className="org.springframework.web.struts.ContextLoaderPlugIn">
        </plug-in>
</struts-config>
<!--action-servlet.xml-->  [Spring Configuration File]
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC  "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
```

```
<bean  name="/demo"   class="demopack.DemoAction" >
        <property  name="model">
                <ref bean="d1" />
        </property>
 </bean>
 <bean   id="d1" class="demopack.ModelImpl" >
</bean>
</beans>
```

// DemoForm.java

```
package demopack;
import org.apache.struts.action.*;
public class DemoForm  extends ActionForm
{
        private  String uname;
        public void setUname(String s)
        {
                uname=s;
        }
        public String getUname()
        {
                return uname;
        }
}
```

//ModelInter.java

```
package demopack;
public interface ModelInter
{
        public  String getWish();
}
```

// ModelImpl.java

```
package demopack;
import java.util.*;
public class ModelImpl implements ModelInter
{
        public  String getWish()
```

```
6041        {
6042                Calendar c1=Calendar.getInstance();
6043                int h=c1.get(Calendar.HOUR_OF_DAY);
6044                if(h  < 12)
6045                return "Good Morning";
6046                else if( h < 17)
6047                return "Good Afternoon";
6048                else
6049                return "Good Evening";
6050        }
6051 }
6052 // DemoAction.java
6053 package demopack;
6054 import javax.servlet.*;
6055 import javax.servlet.http.*;
6056 import java.io.*;
6057 import org.apache.struts.action.*;
6058 public class DemoAction  extends Action
6059 {
6060     private    ModelInter  model;
6061     public void setModel(ModelInter i1)
6062     {
6063                 model=i1;
6064     }
6065     public ActionForward execute(ActionMapping mapping,ActionForm form,HttpServletRequest
6066 request,HttpServletResponse response)
6067     throws ServletException,IOException
6068     {
6069          DemoForm   f1=(DemoForm)form;
6070          String  uname=f1.getUname();
6071           String msg=model.getWish()+" "+ uname;
6072           request.setAttribute("message",msg);

6073         return mapping.findForward("success");
6074      }
```

```
}
<!-- result.jsp -->
<%@  page language="java" %>
<%@ taglib  prefix="html"  uri="struts-html"  %>
<html>
<body bgcolor=#ffffcc text=green>
<h1><center>struts framewrok</h1><hr><br><h3>
<%=  request.getAttribute("message")  %>
<br><br>
<html:link  href="index.jsp" >home</html:link>
</body>
</html>
```

Note: In "lib" folder of WEB_INF we need to place related jar files of spring & struts.

//Example: Integration of struts, spring & hibernate

```
<!-- web.xml -->
<!DOCTYPE web-app
  PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
  "http://java.sun.com/j2ee/dtds/web-app_2_3.dtd">
<web-app>
  <servlet>
        <servlet-name>action</servlet-name>
        <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
        <init-param>
              <param-name>config</param-name>
              <param-value>/WEB-INF/struts-config.xml</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>action</servlet-name>
        <url-pattern> *.do </url-pattern>
    </servlet-mapping>
    <welcome-file-list>
        <welcome-file>index.jsp</welcome-file>
    </welcome-file-list>
```

```
6110        <taglib>
6111            <taglib-uri>struts-html</taglib-uri>
6112            <taglib-location>/WEB-INF/struts-html.tld</taglib-location>
6113        </taglib>
6114    </web-app>
6115    <!-- index.jsp -->
6116    <%@  page language="java" %>
6117    <%@ taglib  prefix="html"  uri="struts-html"  %>
6118    <html>
6119    <body bgcolor=#ffffcc text=green>
6120    <h1><center>new user information</h1><hr><br><h3>
6121    <html:form action="/user">
6122      enter user name :<html:text  property="uname"  /> <br><br>
6123      select role:
6124    <table>
6125    <tr><td>
6126     <html:radio   property="role" value="Administrator">Administrator</html:radio>
6127     <td><html:radio   property="role" value="User">User</html:radio>
6128    </tr>
6129    <tr>
6130     <td><html:radio   property="role" value="Salesman">Salesman</html:radio>
6131    <td>  <html:radio   property="role" value="Manager">Manager</html:radio>
6132    </tr>
6133    </table>
6134      <html:submit  value="save" />
6135    </html:form>
6136    </body>
6137    </html>
6138    <!-- struts-config.xml -->
6139    <?xml version="1.0" encoding="ISO-8859-1" ?>
6140    <!DOCTYPE struts-config PUBLIC "-//Apache Software Foundation//DTD Struts Configuration
6141    1.1//EN"
6142                            "http://jakarta.apache.org/struts/dtds/struts-config_1_1.dtd">
6143    <struts-config>
```

```
    <form-beans>
        <form-bean   name="userform"    type="demo.UserForm" />
     </form-beans>
    <action-mappings>
        <action  path="/user"  name="userform"
        type="org.springframework.web.struts.DelegatingActionProxy" >
            <forward  name="success"  path="/result.jsp" />
            <forward  name="failure"  path="/index.jsp" />
        </action>
    </action-mappings>
    <plug-in className="org.springframework.web.struts.ContextLoaderPlugIn">
    </plug-in>
</struts-config>
<!-- action-servlet.xml-->  [Spring Configuration file]
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC  "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
 <bean  name="/user"   class="demo.UserAction" >
        <property  name="model">
                        <ref bean="d1" />
        </property>
 </bean>
 <bean   id="d1" class="demo.ModelImpl" >
        <property  name="sesfact">
                        <ref bean="mySessionFactory" />
        </property>
 </bean>
 <bean id="myDataSource"
    class="org.springframework.jdbc.datasource.DriverManagerDataSource">
   <property name="driverClassName">
     <value>oracle.jdbc.driver.OracleDriver</value>
   </property>
   <property name="url">
     <value>jdbc:oracle:thin:@localhost:1521:sathya</value>
```

```
    </property>
    <property name="username">
      <value>scott</value>
    </property>
    <property name="password">
      <value>tiger</value>
    </property>
  </bean>
<bean id="mySessionFactory"
class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
<property name="dataSource" ref="myDataSource"/>
<property name="mappingResources">
        <list>
            <value>User.hbm.xml</value>
        </list>
</property>
<property name="hibernateProperties">
    <props>
        <prop key="hibernate.dialect">org.hibernate.dialect.Oracle9Dialect</prop>
     </props>
</property>
</bean>
</beans>
```

//ModelInter.java

```
package demo;
public interface ModelInter
{
        public  boolean insertInfo(String uname,String role);
}
```

// ModelImpl.java

```
package demo;
import java.util.*;
import org.hibernate.*;
public class ModelImpl implements ModelInter
{
```

```
6214        private  SessionFactory sesfact;
6215        public void setSesfact(SessionFactory sf)
6216        {
6217                sesfact=sf;
6218        }
6219        public  boolean insertInfo(String uname,String role)
6220        {           Session ses=sesfact.openSession();
6221                Transaction tx=ses.beginTransaction();
6222                boolean flag=false;

6223                try{

6224                  User u1=new User();
6225                   u1.setUname(uname);
6226                u1.setRole(role);
6227                ses.save(u1);
6228                tx.commit();
6229                flag=true;
6230                }catch(Exception e)
6231                {       tx.rollback();
6232                        System.out.println(e);
6233                }
6234                ses.close();
6235                return flag;
6236        }
6237  }
6238  //User.java
6239  package demo;
6240  public class User{
6241        private int uid;
6242        private String uname,role;
6243        public void setUid(int n)
6244        {
6245                uid=n;
6246        }
```

```
6247            public int getUid(){ return uid; }
6248            public void setUname(String s){ uname=s; }
6249            public String getUname(){ return uname; }
6250            public void setRole(String r){ role=r; }
6251            public String getRole(){  return role; }
6252    }
6253    <!-- User.hbm.xml -->
6254    <?xml version="1.0"?>
6255    <!DOCTYPE hibernate-mapping PUBLIC
6256    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
6257    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
6258    <hibernate-mapping>
6259        <class     name="demo.User"   table="users" >
6260            <id   name="uid"  column="userid"  >
6261                    <generator  class="increment" />
6262            </id>
6263            <property   name="uname" />
6264            <property   name="role" />
6265        </class>
6266    </hibernate-mapping>
6267    // UserForm.java
6268    package demo;
6269    import org.apache.struts.action.*;
6270    public class UserForm  extends ActionForm
6271    {
6272            private  String uname,role;
6273            public void setUname(String s)
6274            {
6275                    uname=s;
6276            }
6277            public String getUname()
6278            {
6279                    return uname;
6280            }
6281            public void setRole(String s){ role=s; }
```

```
6282            public String getRole(){ return role; }
6283
6284    }
6285    // UserAction.java
6286    package demo;
6287    import javax.servlet.*;
6288    import javax.servlet.http.*;
6289    import java.io.*;
6290    import org.apache.struts.action.*;
6291    public class UserAction  extends Action
6292    {    private    ModelInter  model;
6293         public void setModel(ModelInter i1)
6294         {
6295               model=i1;
6296          }
6297         public ActionForward execute(ActionMapping mapping,ActionForm form,HttpServletRequest
6298    request,HttpServletResponse response)
6299         throws ServletException,IOException
6300         {
6301               UserForm   f1=(UserForm)form;
6302                 String uname=f1.getUname();
6303                  String role=f1.getRole();
6304                if(model.insertInfo(uname,role))
6305                        return mapping.findForward("success");
6306              else
6307                        return mapping.findForward("failure");
6308          }
6309    }
6310    <!-- result.jsp -->
6311    <%@  page language="java" %>
6312    <%@ taglib  prefix="html"  uri="struts-html"  %>
6313    <html>  <body bgcolor=#ffffcc text=green>
6314    <h1><center>welcome user</h1><hr>
6315    </body>
6316    </html>
```