PART-1

Spring-IOC

Spring-DAO

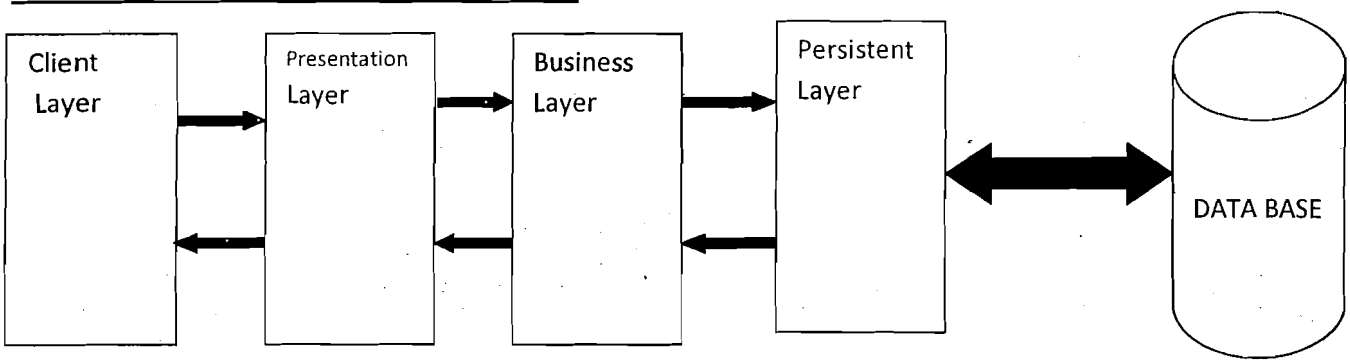
Q) Where actually java is used?

- ⇨ Java is used to develop the enterprise applications.
- ⇨ Enterprise means business organization
- ⇨ Business organization provides services

Q) What is Enterprise Application?

- ⇨ Computerizing business applications

Architecture of Enterprise Application:



Client Layer:

- ➢ It is browser software.

Presentatation Layer:

Roles:

- ➢ Receiving user request from client tier (calling request)
- ➢ Capturing the user provided data
- ➢ Validating the user input
- ➢ Calls the business method to get business services
- ➢ Keep the processed data in memory.
- ➢ Forwarding the control to output jsp

Note: should not write business logic in the presentation tier

Business Layer:

- ➢ Receiving request from presentation tier
- ➢ Contacting the persistent tier to get the database data
- ➢ Implementing the business logic
- ➢ Return the control to presentation tlier

Persistent Layer:

- ➢ Receiving the request from business tier
- ➢ Contacting database to get the database data
- ➢ Return the accessed data to business tier

Data Layer:

- ➢ It is a database.

Q) What are the different logics available in Enterprise Application?

- **Presentation Logic:** This logic generally we will write in JSP.
- **Application/Controlling Logic:** Generally we write this logic in controller.
- **Bussiness Logic:** Programmatical implementation of business rules is nothing but bussiness rules.
- **DataAccess Logic:** The logic which will we write to contact the DB.

Q) What are the Sun Microsystems technologies and frameworks in enterprise application development?

Presentation tier	Business tier	Persistent tier
Servlets jsp's jsf	EJB2 session beans EJB3 session beans MDB(Message Driven Beans)	JDBC JPA(java persistent API)

Q) What are the non-Sun Microsystems technologies and frameworks in enterprise application development?

Presentation tier	Business tier	Persistent tier
Struts Velocity Freemarker Spring Web MVC	Spring Aop Spring JEE	Hibernate Ibatis Toplink JDO Spring DAO Spring ORM

- So spring is used to develop any layer of the Enterprise Application.

Note:

- ⇨ Spring is not a technology from sun micro system like JDBC, EJB, JSP, servlets...etc
- ⇨ Spring is neither J2SE nor j2EE but it uses both

Q) What is spring actually?

- ⇨ It is a open source frame work
- ⇨ Spring is a light-weight framework
- ⇨ Industry treats spring as framework of framework.

Note: spring is very much successful in the industry in business tier development as an alternate to EJB2. Later on it is used in all layers of the application because of its light weightness.

Q) What is Framework?

- ⇨ It is a reusable semi-finished application that can be customized according to our business requirement

Q) What is light-weight in context of spring?

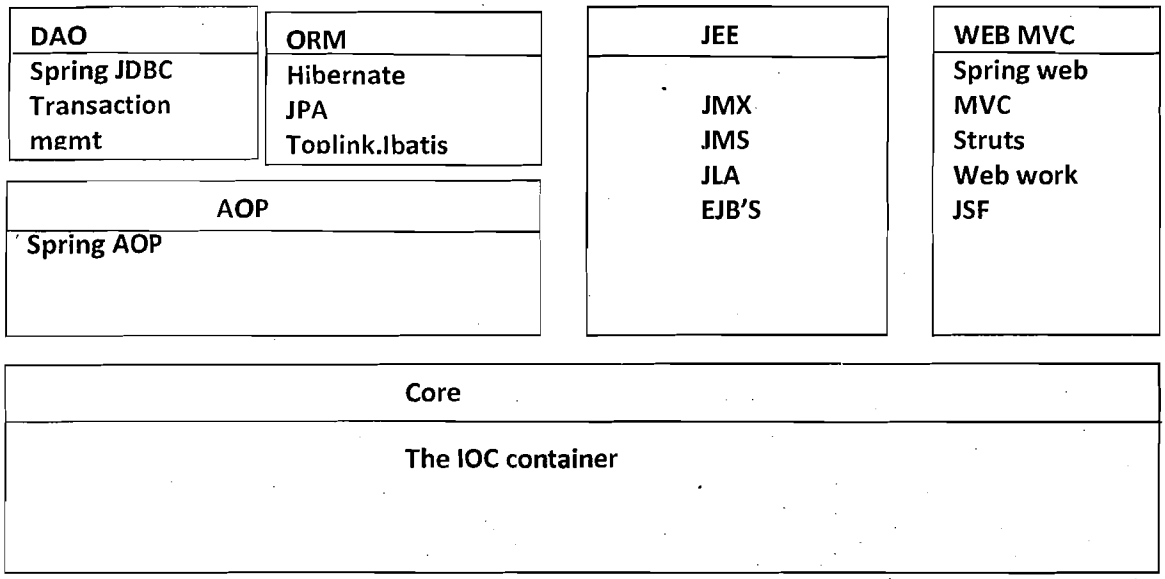
It is light-weight in terms of

1. Size of container
2. Processing over heads involved in giving the enterprise services
3. Business class dependency & amount of coupling with the framework

Q) What was the main objective of spring invention?

- ⇨ Simplification of j2ee application by reducing the complexity involved in enterprise application development.

Spring Architecture



1) Core module:
- ⇨ The core module of spring framework is spring container. Spring beans are managed by spring container.
- ⇨ This module concentrates on major principles of IOC (Inversion Of Control)
- ⇨ We call IOC as heart of spring applications
- ⇨ IOC is a container used for holding spring beans

2) DAO module:
- ⇨ It is meant for building persistence tier (data access layer)
- ⇨ In this framework it provides JDBC abstraction

3) ORM:
- ⇨ Spring provides an integration support with most of the ORM tools

Ex: Hibernate, top Link, JPA, Ibatis...etc

4) JEE module:
- ⇨ This module provides the SUN enterprise services to the spring bean
- ⇨ In this module spring has the support for integrating with EJB's, RMI, JAVA Mail, Transactions(JTA,JTS)...etc

5) web MVC module:
- ⇨ This module is used to develop presentation tier of an Enterprise Application.
- ⇨ Spring has an implementation for the MVC design pattern we call it as spring MVC module
- ⇨ Spring also provides an integration with any MVC implementations like struts,JSF...etc

6) AOP module:
- ⇨ To solve the cross cutting concern problems, spring using Aspect Oriented Programming(AOP)
- ⇨ To provide declarative enterprise services (logging, security, transaction management...etc) AOP is used.
- ⇨ Using AOP we can integrate other AOP framework also.
  Ex: aspectJ

Design pattern: solutions for repeated problems .
Ex: SingleTon, Factory, MVC, business delegates, service locator..etc

ConnectionUtilizer001.java

```
1. public class ConnectionUtilizer001 {
2.     public static void main(String[] args) {
3.           System.out.println("this is Oracle Connection");
4.     }
5. }
```

⇨ In the above Utilizer class if we want Mysql Connection, then we need to modify the above program as follows.

ConnectionUtilizer002.java

```
1. public class ConnectionUtilizer002 {
2.     public static void main(String[] args) {
3.           System.out.println("this is Mysql Connection");
4.     }
5. }
```

⇨ So when we want to get the services of other provider, we are modifying our utilizer class.
⇨ And in the above code we are mixing Utilizer code and Provider code. Means Utilizer is tightly coupled with one particular provider. So, it is advisable to write different classes for Utilizer logic and Provider logic.

OracleConnectionProvider.java

```
1. public class OracleConnectionProvider {
2.     public String getOracleConnection() {
3.           return "this is oracle connection";
4.     }
5. }
```

MysqlConnectionProvider.java

```
1. public class MysqlConnectionProvider {
2.     public String getMysqlConnection() {
3.           return "this is mysql connection";
4.     }
5. }
```

ConnectionUtilizer003.java

```
1. public class ConnectionUtilizer003 {
2.     public static void main(String[] args) {
3.           OracleConnectionProvider oracleProvider = new
4.                                                 OracleConnectionProvider ();
5.           String connectionString = oracleProvider.getOracleConnection();
6.           System.out.println(connectionString);
7.     }
8. }
```

⇨ If we want to get Mysql Connection in the above example, we need to create that particular provider object in our class(MysqlConnectionProvider.java). So, we have to modify the above program as follows.

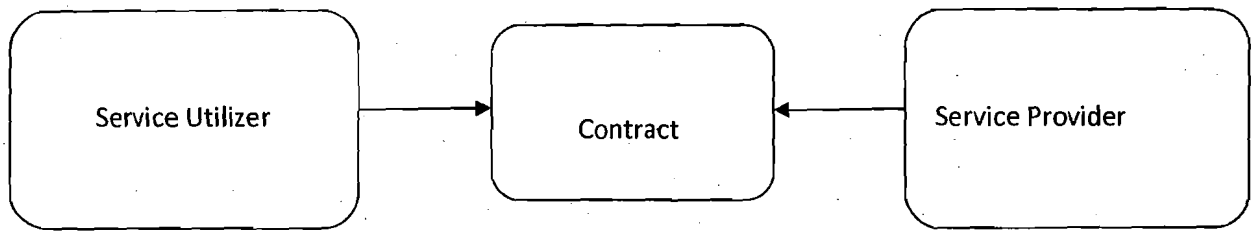
ConnectionUtilizer004.java

```
1. public class ConnectionUtilizer004 {
2.     public static void main(String[] args) {
3.           MysqlConnectionProvider mysqlProvider = new
4.                                               MysqlConnectionProvider ();
5.           String connectionString = mysqlProvider.getMysqlConnection();
6.           System.out.println(connectionString);
7.     }
8. }
```

⇨ If we observe **ConnectionUtilizer003, ConnectionUtilizer004** The way we are getting provider services is changing when the provider is changing. Means the way we are calling methods are changing when provider is changing.
  - To get Mysql Connection we are calling connection.getMysqlConnection();
  - To get Oracle Connection we are calling connection.getOracleConnection();

⇨ To solve the above problem we should have a contract between service provider and service utilize.

⇨ In java we can provide the contract with **interfaces.**



⇨ So the above program can be refactor as follows.

Connection.java

```
1. public interface Connection {
2.     public String getConnection();
3. }
```

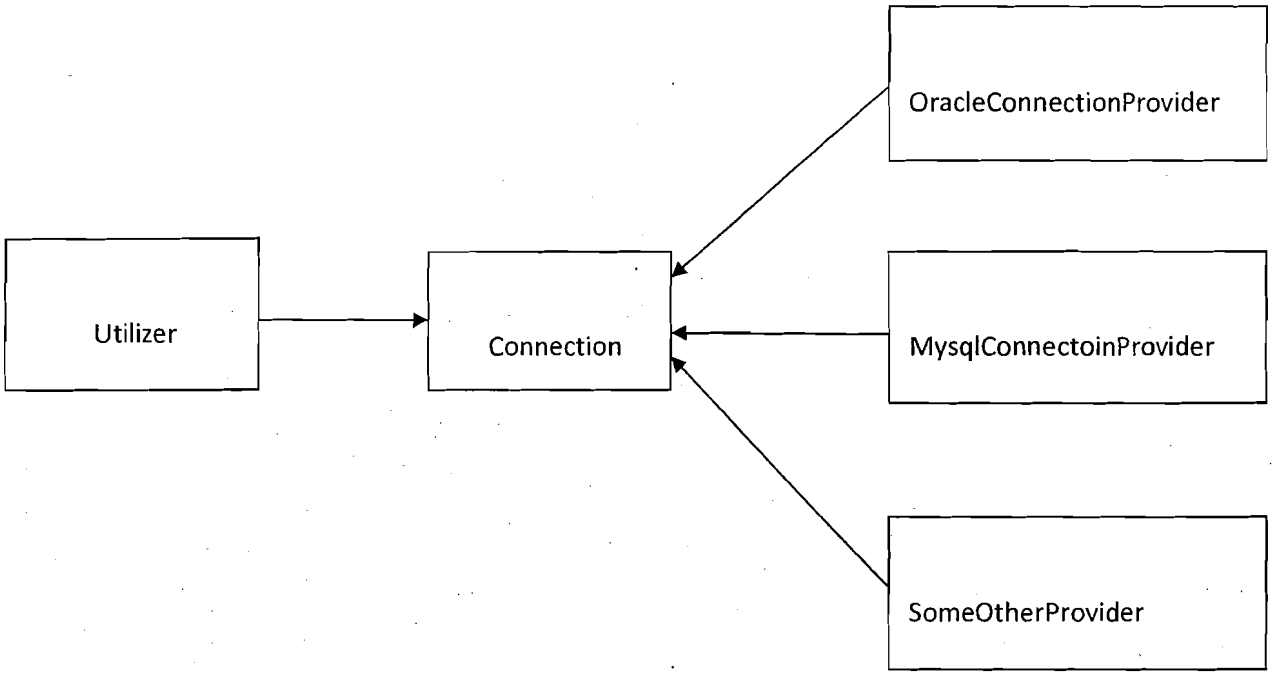
OracleConnectionProvider.java

```
1. public class OracleConnectionProvider implements Connection {
2.     public String getConnection() {
3.           return "this is oracle connection";
4.     }
```

```
5. }
```

MysqlConnectionProvider.java

```
1. public class MysqlConnectionProvider implements Connection {
2.
3.     public String getConnection() {
4.           return "this is mysql connection";
5.     }
6. }
```



ConnectionUtilizer005.java

```
1. public class ConnectionUtilizer005 {
2. public static void main(String[] args) {
3.     Connection contract = new OracleConnectionProvider();
4.     String connectionString = contract.getConnection();
5.     System.out.println(connectionString);
6. }
7. }
```

ConnectionUtilizer006.java

```
1. public class ConnectionUtilizer006 {
2. public static void main(String[] args) {
3.     Connection contract = new MysqlConnectionProvider();
4.     String connectionString = contract.getConnection();
5.     System.out.println(connectionString);
6. }
7. }
```

- ⇨ If we observe **ConnectionUtilizer005, ConnectionUtilizer006** The way we are getting provider services is not changing even, when the provider is changing. Means the way we are calling methods are not changing when provider is changing.
  - To get Mysql Connection we are calling contract.getConnection();
  - To get Oracle Connection we are calling contract.getConnection();

- ⇨ Now if we observe **ConnectionUtilizer005, ConnectionUtilizer006** when the provider changing, the we are creating that particular object in the utilizer class.
- ⇨ So, when the provider is changing we are modifying only one line of code(where we are creating particular provider object).
- ⇨ As we know even if we change one line of code, we need to recompile the application. So, when we change the provider class we need to re-compile the application. So our Utilizer class is always need to change when the provider is changing.
- ⇨ So we need to get the Provider object dynamically.
- ⇨ So we need to specify our required provider declaratively.
- ⇨ To specify our requirements declaratively we use either properties file or xml file.
- ⇨ In this example we specify, our requirements through properties file.
- ⇨ We will write one container which will read our requirements and provide our required objects dynamically.

Connection.java

```
1. public interface Connection {
2.     public String getConnection();
3. }
```

OracleConnectionProvider.java

```
1. public class OracleConnectionProvider implements Connection {
2.
3.     public String getConnection() {
4.           return "this is oracle connection";
5.     }
6. }
```

MysqlConnectionProvider.java

```
1. public class MysqlConnectionProvider implements Connection {
2.
3.     public String getConnection() {
4.           return "this is mysql connection";
5.     }
6. }
```

myproperties.properties

```
1. provider=OracleConnectionProvider
```

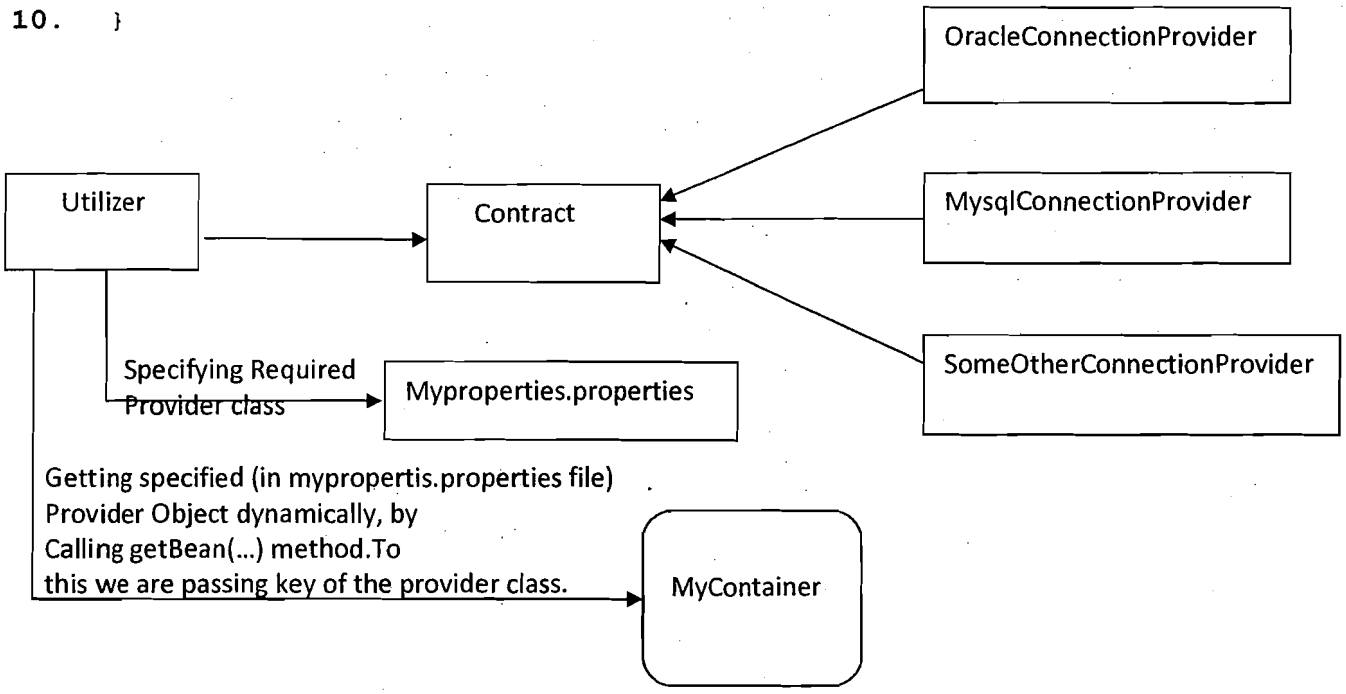
MyContainer.java

```
1. import java.io.IOException;
2. import java.util.Properties;
3.
4. public class MyContainer {
5.     private static Properties properties;
6.     static {
7.           try {
8.                 properties = new Properties();
9.                 properties.load(MyContainer.class.getClassLoader()
10.                       .getResourceAsStream("myproperties.properties"));
11.                } catch (IOException e) {
12.                      e.printStackTrace();
13.                }
14.          }
15.
```

```
16.         public static Object getBean(String providerKey) {
17.               Object object = null;
18.               String providerClassName =
19.                                       properties.getProperty(providerKey);
20.               try {
21.                     Class c = Class.forName(providerClassName);
22.                     object = c.newInstance();
23.               } catch (Exception e) {
24.                     e.printStackTrace();
25.               }
26.               return object;
27.         }
28.   }
```

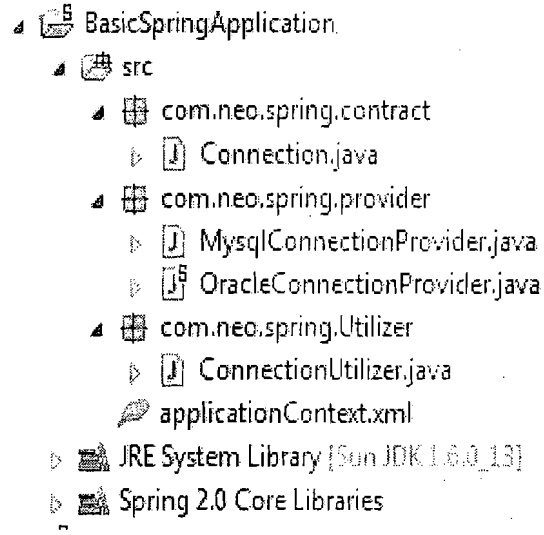
ConnectionUtilizer007.java

```
1. public class ConnectionUtilizer007 {
2.
3.    private static MyContainer container = new MyContainer();
4.
5.    public static void main(String[] args) {
6.          Connection contract = (Connection) container.getBean("provider");
7.          String connectionString = contract.getConnection();
8.          System.out.println(connectionString);
9.    }
10.   }
```



- ⇨ Now if you want to change the provider class, just change the properties file. We no need to change the Utilizer class.
- ⇨ Here we are implementing the container code. But in the case of spring, Spring provides the container.

First Example on Spring



Connection.java

```
1. package com.neo.spring.contract;
2.
3. public interface Connection {
4.
5.    public String getConnection();
6.
7. }
```

MysqlConnectionProvider.java

```
1. package com.neo.spring.provider;
2.
3. import com.neo.spring.contract.Connection;
4.
5. public class MysqlConnectionProvider implements Connection {
6.
7.    public String getConnection() {
8.          return "this is mysql connection";
9.    }
10.   }
```

OracleConnectionProvider.java

```
1. package com.neo.spring.provider;
2.
3. import com.neo.spring.contract.Connection;
4.
5. public class OracleConnectionProvider implements Connection {
6.
7.    public String getConnection() {
8.          return "this is oracle connection";
9.    }
10.   }
```

ConnectionUtilizer.java

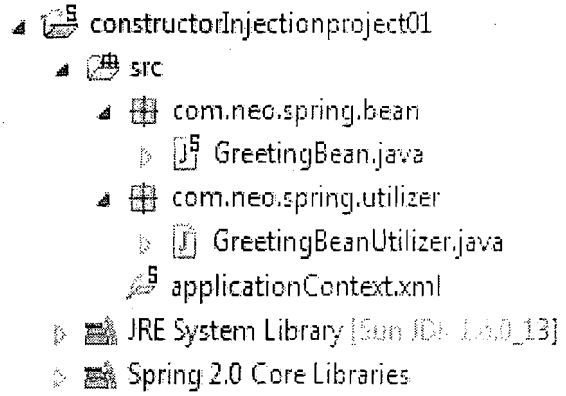
```
1. package com.neo.spring.Utilizer;
2.
```

```
3. import org.springframework.beans.factory.BeanFactory;
4. import org.springframework.beans.factory.xml.XmlBeanFactory;
5. import org.springframework.core.io.ClassPathResource;
6.
7. import com.neo.spring.contract.Connection;
8.
9. public class ConnectionUtilizer {
10.         private static BeanFactory factory = new XmlBeanFactory(
11.                     new ClassPathResource("applicationContext.xml"));
12.
13.         public static void main(String[] args) {
14.
15.               Connection contract = (Connection)
16.                                       factory.getBean("provider");
17.               String connectionstring = contract.getConnection();
18.               System.out.println(connectionstring);
19.         }
20.   }
```

applicationContext.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans
3.    xmlns="http://www.springframework.org/schema/beans"
4.    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5.    xsi:schemaLocation="http://www.springframework.org/schema/beans
6. http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">
7.     <!--<bean id="provider"
8.         class="com.neo.spring.provider.MysqlConnectionProvider"/>-->
9.     <bean id="provider"
10.          class="com.neo.spring.provider.OracleConnectionProvider"/>
11.
12.   </beans>
```

Primitive Datatype Constructor Injection



GreetingBean.java

```
1. package com.neo.spring.bean;
2.
3. public class GreetingBean {
4.     private String name;
5.
6.     public GreetingBean() {
7.     }
8.
9.     GreetingBean(String name) {
10.                this.name = name;
11.        }
12.
13.        public String getName() {
14.                return name;
15.        }
16.
17.        public void setName(String name) {
18.                this.name = name;
19.        }
20.
21.    }
```

GreetingBeanUtilizer.java

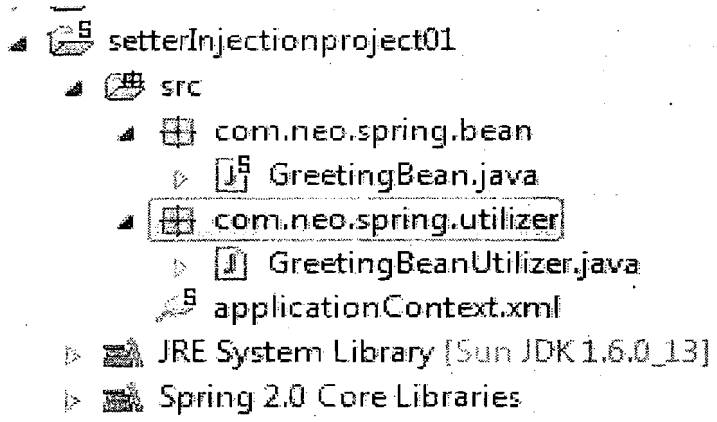
```
1. package com.neo.spring.utilizer;
2.
3. import org.springframework.beans.factory.BeanFactory;
4. import org.springframework.beans.factory.xml.XmlBeanFactory;
5. import org.springframework.core.io.ClassPathResource;
6.
7. import com.neo.spring.bean.GreetingBean;
8.
9. public class GreetingBeanUtilizer{
10.        private static BeanFactory factory = new XmlBeanFactory(
11.                        new ClassPathResource("applicationContext.xml"));
12.
13.        public static void main(String[] args) {
14.                GreetingBean bean = (GreetingBean) factory.getBean("gb");
15.                String name = bean.getName();
16.                System.out.println("Welcome to spring world : " + name);
17.
18.        }
```

```
19.
20.   }
```

applicationContext.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3.    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.    xsi:schemaLocation="http://www.springframework.org/schema/beans
5. http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">
6.    <bean id="gb" class="com.neo.spring.bean.GreetingBean">
7.          <constructor-arg value="SOMASEKHAR" />
8.
9.    </bean>
10.
11.   </beans>
```

Primitive Datatype Setter Injection



GreetingBean.java

```
1. package com.neo.spring.bean;
2.
3. public class GreetingBean {
4.    private String name;
5.
6.    GreetingBean() {
7.    }
8.
9.    GreetingBean(String name) {
10.             this.name = name;
11.       }
12.
13.       public String getName() {
14.             return name;
15.       }

16.       public void setName(String name) {
17.             this.name = name;
18.       }
19.
20.   }
```

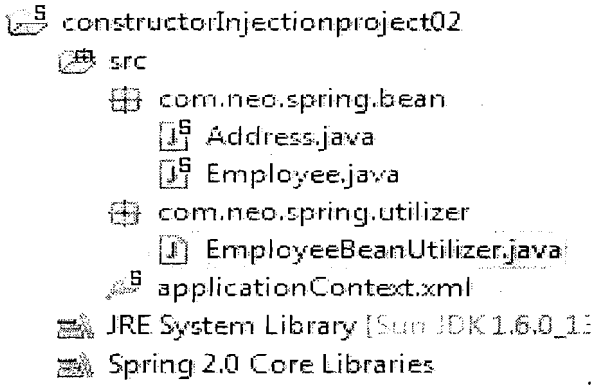
GreetingBeanUtilizer.java

```
1. package com.neo.spring.utilizer;
2.
3. import org.springframework.beans.factory.BeanFactory;
4. import org.springframework.beans.factory.xml.XmlBeanFactory;
5. import org.springframework.core.io.ClassPathResource;
6.
7. import com.neo.spring.bean.GreetingBean;
8.
9. public class GreetingBeanUtilizer{
10.         private static BeanFactory factory = new XmlBeanFactory(
11.                     new ClassPathResource("applicationContext.xml"));
12.
13.         public static void main(String[] args) {
14.
15.               GreetingBean bean = (GreetingBean) factory.getBean("gb");
16.               String name = bean.getName();
17.               System.out.println("Welcome to spring world: "+name);
18.
19.         }
20.
21.   }
```

applicationContext.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3.     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.     xsi:schemaLocation="http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">
5.     <bean id="gb" class="com.neo.spring.bean.GreetingBean">
6.           <property name="name" value="sekhar"></property>
7.
8.     </bean>
9. </beans>
```

Object type Constructor Injection



Address.java

```
1. package com.neo.spring.bean;
2.
3. public class Address {
4.     private int hno;
5.     private String city;
6.
7.     public Address() {
8.     }
9.
10.         public Address(int hno, String city) {
11.               this.hno = hno;
12.               this.city = city;
13.         }
14.
15.         public int getHno() {
16.               return hno;
17.         }
18.
19.         public void setHno(int hno) {
20.               this.hno = hno;
21.         }
22.
23.         public String getCity() {
24.               return city;
25.         }
26.
27.         public void setCity(String city) {
28.               this.city = city;
29.         }
30.
31.   }
```

Employee.java

```
1. package com.neo.spring.bean;
2.
3. public class Employee {
4.     private int eno;
5.     private String ename;
6.     private Address address;
7.
8.     public Employee() {
```

```
9.    }
10.
11.         public Employee(int eno, String ename, Address address) {
12.               this.eno = eno;
13.               this.ename = ename;
14.               this.address = address;
15.         }
16.
17.         public int getEno() {
18.               return eno;
19.         }
20.
21.         public void setEno(int eno) {
22.               this.eno = eno;
23.         }
24.
25.         public String getEname() {
26.               return ename;
27.         }
28.
29.         public void setEname(String ename) {
30.               this.ename = ename;
31.         }
32.
33.         public Address getAddress() {
34.               return address;
35.         }
36.
37.         public void setAddress(Address address) {
38.               this.address = address;
39.         }
40.   }
```

EmployeeBeanUtilizer.java

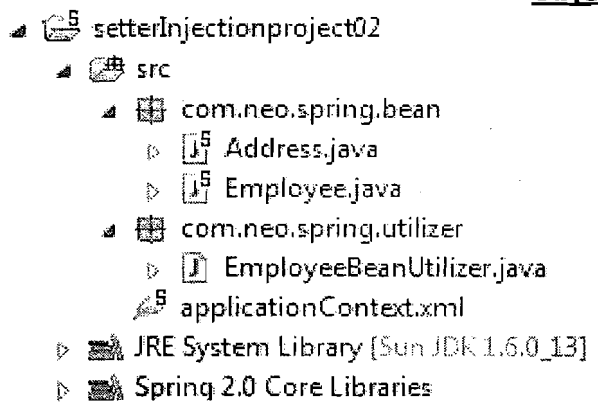
```
1. package com.neo.spring.utilizer;
2.
3. import org.springframework.beans.factory.BeanFactory;
4. import org.springframework.beans.factory.xml.XmlBeanFactory;
5. import org.springframework.core.io.ClassPathResource;
6.
7. import com.neo.spring.bean.Address;
8. import com.neo.spring.bean.Employee;
9.
10.   public class EmployeeBeanUtilizer{
11.         private static BeanFactory factory = new XmlBeanFactory(
12.                     new ClassPathResource("applicationContext.xml"));
13.
14.         public static void main(String[] args) {
15.               System.out.println("Employee details...");
16.               Employee employee = (Employee) factory.getBean("employee");
17.               System.out.println(employee.getEno());
18.               System.out.println(employee.getEname());
19.               Address address = employee.getAddress();
20.               System.out.println("Employee address details...");
21.               System.out.println(address.getHno());
```

```
22.                 System.out.println(address.getCity());
23.           }
24.     }
```

applicationContext.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3.     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.     xsi:schemaLocation="http://www.springframework.org/schema/beans
5. http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">
6.
7.     <bean id="addr" class="com.neo.spring.bean.Address">
8.           <constructor-arg value="1702" />
9.           <constructor-arg value="Hyderabad" />
10.          </bean>
11.          <bean id="employee" class="com.neo.spring.bean.Employee">
12.                <constructor-arg value="88688" />
13.                <constructor-arg value="sekhar" />
14.                <constructor-arg ref="addr" />
15.          </bean>
16.
17.    </beans>
```

Object type Setter Injection



Address.java

```
1. package com.neo.spring.bean;
2.
3. public class Address {
4.     private int hno;
5.     private String city;
6.
7.     public Address() {
8.     }
9.
10.          public Address(int hno, String city) {
11.                this.hno = hno;
12.                this.city = city;
13.          }
14.
15.          public int getHno() {
16.                return hno;
17.          }
```

```
18.
19.         public void setHno(int hno) {
20.               this.hno = hno;
21.         }
22.
23.         public String getCity() {
24.               return city;
25.         }
26.
27.         public void setCity(String city) {
28.               this.city = city;
29.         }
30.   }
```

Employee.java

```
1. package com.neo.spring.bean;
2.
3. public class Employee {
4.    private int eno;
5.    private String ename;
6.    private Address address;
7.
8.    public Employee() {
9.    }
10.
11.         public Employee(int eno, String ename, Address address) {
12.               this.eno = eno;
13.               this.ename = ename;
14.               this.address = address;
15.         }
16.
17.         public int getEno() {
18.               return eno;
19.         }
20.
21.         public void setEno(int eno) {
22.               this.eno = eno;
23.         }
24.
25.         public String getEname() {
26.               return ename;
27.         }
28.
29.         public void setEname(String ename) {
30.               this.ename = ename;
31.         }
32.
33.         public Address getAddress() {
34.               return address;
35.         }
36.
37.         public void setAddress(Address address) {
38.               this.address = address;
39.         }
```

```
40.
41.   }
42.
```

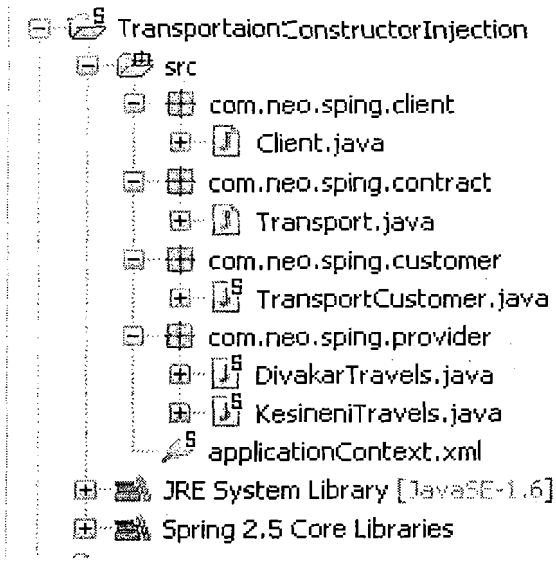
EmployeeBeanUtilizer.java

```
1. package com.neo.spring.utilizer;
2.
3. import org.springframework.beans.factory.BeanFactory;
4. import org.springframework.beans.factory.xml.XmlBeanFactory;
5. import org.springframework.core.io.ClassPathResource;
6.
7. import com.neo.spring.bean.Address;
8. import com.neo.spring.bean.Employee;
9.
10.   public class EmployeeBeanUtilizer{
11.         private static BeanFactory factory = new XmlBeanFactory(
12.                     new ClassPathResource("applicationContext.xml"));
13.
14.         public static void main(String[] args) {
15.               System.out.println("Employee details...");
16.               Employee employee = (Employee) factory.getBean("employee");
17.               System.out.println(employee.getEno());
18.               System.out.println(employee.getEname());
19.               Address address = employee.getAddress();
20.               System.out.println("Employee address details...");
21.               System.out.println(address.getHno());
22.               System.out.println(address.getCity());
23.         }
24.
25.   }
26.
```

applicationContext.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3.    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.    xsi:schemaLocation="http://www.springframework.org/schema/beans
5. http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">
6.
7.    <bean id="addr" class="com.neo.spring.bean.Address">
8.          <property name="hno" value="18795"></property>
9.          <property name="city" value="Hyderabad"></property>
10.         </bean>
11.         <bean id="employee" class="com.neo.spring.bean.Employee">
12.               <property name="eno" value="101"></property>
13.               <property name="ename" value="sekhar"></property>
14.               <property name="address" ref="addr"></property>
15.         </bean>
16.   </beans>
```

IOC Example



Client.java

```
1. package com.neo.sping.client;
2.
3. import org.springframework.beans.factory.BeanFactory;
4. import org.springframework.beans.factory.xml.XmlBeanFactory;
5. import org.springframework.core.io.ClassPathResource;
6.
7. import com.neo.sping.customer.TransportCustomer;
8.
9. public class Client {
10.         private static BeanFactory factory = new XmlBeanFactory(
11.                     new ClassPathResource("applicationContext.xml"));
12.
13.         public static void main(String[] args) {
14.               TransportCustomer customer = (TransportCustomer)
15.                                                   factory.getBean("tc");
16.               customer.useTransport();
17.         }
18.   }
```

Transport.java

```
1. package com.neo.sping.contract;
2.
3. public interface Transport {
4.    void doTravel();
5. }
```

TransportCustomer.java

```
1. package com.neo.sping.customer;
2.
3. import com.neo.sping.contract.Transport;
4.
5. public class TransportCustomer {
6.
```

```
7.    private Transport transport;
8.
9.    public TransportCustomer() {
10.
11.         }
12.
13.         public TransportCustomer(Transport transport) {
14.               this.transport = transport;
15.         }
16.
17.         public void setTransport(Transport transport) {
18.               this.transport = transport;
19.         }
20.
21.         public void useTransport() {
22.               transport.doTravel();
23.         }
24.   }
```

DivakarTravels.java

```
1. package com.neo.sping.provider;
2.
3. import com.neo.sping.contract.Transport;
4.
5. public class DivakarTravels implements Transport {
6.    public DivakarTravels() {
7.
8.    }
9.
10.   public void doTravel() {
11.         System.out.println("this is best DivakarTravels, happy journy");
12.         }
13.   }
```

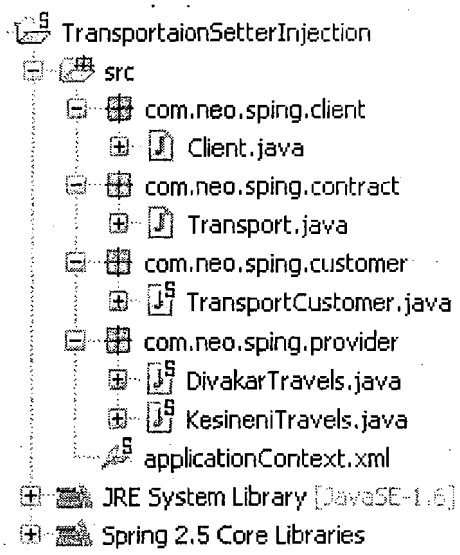
KesineniTravels.java

```
1. package com.neo.sping.provider;
2.
3. import com.neo.sping.contract.Transport;
4.
5. public class KesineniTravels implements Transport {
6.    public KesineniTravels() {
7.    }
8.
9.    public void doTravel() {
10.         System.out.println("this is very best KesineniTravels, have a
11.                                                          nice journey");
12.         }
13.   }
```

applicationContext.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3.    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xmlns:p="http://www.springframework.org/schema/p"
5.    xsi:schemaLocation="http://www.springframework.org/schema/beans
```

```
6. http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
7.
8.     <bean id="dt" class="com.neo.sping.provider.DivakarTravels"></bean>
9.     <bean id="kt" class="com.neo.sping.provider.KesineniTravels"></bean>
10.
11.         <bean id="tc" class="com.neo.sping.customer.TransportCustomer">
12.               <constructor-org  ref="dt"/>
13.         </bean>
14.   </beans>
```



Client.java

```
1. package com.neo.sping.client;
2.
3. import org.springframework.beans.factory.BeanFactory;
4. import org.springframework.beans.factory.xml.XmlBeanFactory;
5. import org.springframework.core.io.ClassPathResource;
6. import com.neo.sping.customer.TransportCustomer;
7.
8. public class Client {
9.     private static BeanFactory factory = new XmlBeanFactory(
10.                       new ClassPathResource("applicationContext.xml"));
11.
12.         public static void main(String[] args) {
13.               TransportCustomer customer = (TransportCustomer)
14.                                                   factory.getBean("tc");
15.               customer.useTransport();
16.         }
17.   }
```

Transport.java

```
1. package com.neo.sping.contract;
2.
3. public interface Transport {
4.     void doTravel();
5. }
```

TransportCustomer.java

```
1. package com.neo.sping.customer;
2.
3. import com.neo.sping.contract.Transport;
4.
5. public class TransportCustomer {
6.
7.     private Transport transport;
8.
9.     public TransportCustomer() {
10.
11.         }
12.
13.         public TransportCustomer(Transport transport) {
14.               this.transport = transport;
15.         }
16.
17.         public void setTransport(Transport transport) {
18.               this.transport = transport;
19.         }
20.
21.         public void useTransport() {
22.               transport.doTravel();
23.         }
24.   }
```

DivakarTravels.java

```
1. package com.neo.sping.provider;
2.
3. import com.neo.sping.contract.Transport;
4.
5. public class DivakarTravels implements Transport {
6.     public DivakarTravels() {
7.
8.     }
9.
10.         public void doTravel() {
11.               System.out.println("this is best DivakarTravels, happy
12.                                                                 journy");
13.         }
14.   }
```

KesineniTravels.java

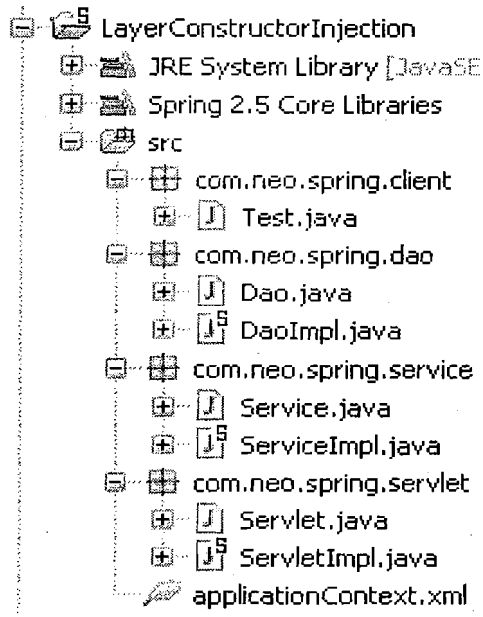
```
1. package com.neo.sping.provider;
2.
3. import com.neo.sping.contract.Transport;
4.
5. public class KesineniTravels implements Transport {
6.     public KesineniTravels() {
7.     }
8.
9.     public void doTravel() {
10.         System.out.println("this is very best KesineniTravels, have a
11.                                                       nice journey");
12.         }
```

```
13.   }
```

applicationContext.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3.    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xmlns:p="http://www.springframework.org/schema/p"
5.    xsi:schemaLocation="http://www.springframework.org/schema/beans
6. http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
7.
8.    <bean id="dt" class="com.neo.sping.provider.DivakarTravels"></bean>
9.    <bean id="kt" class="com.neo.sping.provider.KesineniTravels"></bean>
10.
11.         <bean id="tc" class="com.neo.sping.customer.TransportCustomer">
12.               <property name="transport" ref="dt"></property>
13.         </bean>
14.   </beans>
```

LayerConstructionInjection



Test.java

```
1. package com.neo.spring.client;
2.
3. import org.springframework.beans.factory.BeanFactory;
4. import org.springframework.beans.factory.xml.XmlBeanFactory;
5. import org.springframework.core.io.ClassPathResource;
6.
7. import com.neo.spring.servlet.Servlet;
8.
9. public class Test {
10.         private static BeanFactory factory = new XmlBeanFactory(
11.                     new ClassPathResource("applicationContext.xml"));
12.
13.         public static void main(String[] args) {
14.               Servlet servlet = (Servlet) factory.getBean("servletref");
15.               servlet.servletMethod();
16.         }
17.
18.   }
```

Dao.java

```
1. package com.neo.spring.dao;
2.
3. public interface Dao {
4.      public void daoMethod();
5.
6. }
```

DaoImpl.java

```
1. package com.neo.spring.dao;
2.
3. public class DaoImpl implements Dao{
4.    public DaoImpl() {
```

```
5.    }
6.    public void daoMethod() {
7.        System.out.println("This is DAO layer");
8.    }
9. }
```

Service.java

```
1. package com.neo.spring.service;
2.
3. public interface Service {
4.       public void serviceMethod();
5. }
```

ServiceImpl.java

```
1. package com.neo.spring.service;
2.
3. import com.neo.spring.dao.Dao;
4.
5. public class ServiceImpl implements Service {
6.    private Dao dao;
7.    public ServiceImpl() {
8.    }
9.    public ServiceImpl(Dao dao) {
10.             this.dao=dao;
11.       }
12.       public void setDao(Dao dao) {
13.             this.dao = dao;
14.       }
15.
16.       public void serviceMethod() {
17.             System.out.println("This is Service Layer");
18.             dao.daoMethod();
19.       }
20.   }
```

Servlet.java

```
1. package com.neo.spring.servlet;
2.
3. public interface Servlet {
4.      public void servletMethod();
5. }
```

ServletImpl.java

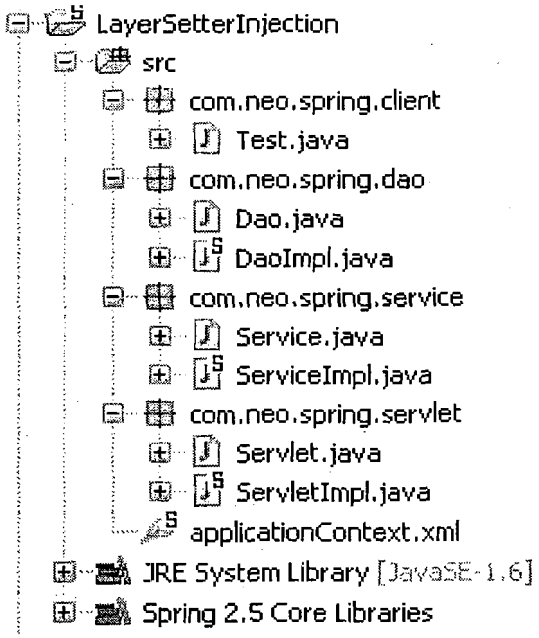
```
1. package com.neo.spring.servlet;
2.
3. import com.neo.spring.service.Service;
4.
5. public class ServletImpl implements Servlet {
6.    private Service service;
7.    public ServletImpl() {
8.    }
9.    public ServletImpl(Service service) {
10.             this.service=service;
11.       }
12.       public void setService(Service service) {
```

```
13.               this.service = service;
14.         }
15.         public void servletMethod() {
16.               System.out.println("This is presentaion layer");
17.               service.serviceMethod();
18.         }
19.
20.   }
```

applicationContext.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3.     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.        xmlns:p="http://www.springframework.org/schema/p"
5.     xsi:schemaLocation="http://www.springframework.org/schema/beans
6.       http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
7.
8.     <bean id="daoref" class="com.neo.spring.dao.DaoImpl">
9.     </bean>
10.          <bean id="serviceref" class="com.neo.spring.service.ServiceImpl">
11.               <constructor-arg ref="daoref"></constructor-arg>
12.          </bean>
13.          <bean id="servletref" class="com.neo.spring.servlet.ServletImpl">
14.               <constructor-arg ref="serviceref"></constructor-arg>
15.
16.          </bean>
17.
18.   </beans>
```

LayerSetterInjection



Test.java

```
1. package com.neo.spring.client;
2.
3. import org.springframework.beans.factory.BeanFactory;
```

```
4. import org.springframework.beans.factory.xml.XmlBeanFactory;
5. import org.springframework.core.io.ClassPathResource;
6.
7. import com.neo.spring.servlet.Servlet;
8.
9. public class Test {
10.         private static BeanFactory factory = new XmlBeanFactory(
11.                     new ClassPathResource("applicationContext.xml"));
12.
13.         public static void main(String[] args) {
14.               Servlet servlet = (Servlet) factory.getBean("servletref");
15.               servlet.servletMethod();
16.         }
17.
18.   }
```

Dao.java

```
1. package com.neo.spring.dao;
2.
3. public interface Dao {
4.      public void daoMethod();
5.
6. }
```

DaoImpl.java

```
1. package com.neo.spring.dao;
2.
3. public class DaoImpl implements Dao{
4.    public DaoImpl() {
5.    }
6.    public void daoMethod() {
7.        System.out.println("This is DAO layer");
8.    }
9. }
```

Service.java

```
1. package com.neo.spring.service;
2.
3. public interface Service {
4.       public void serviceMethod();
5. }
```

ServiceImpl.java

```
1. package com.neo.spring.service;
2.
3. import com.neo.spring.dao.Dao;
4.
5. public class ServiceImpl implements Service {
6.    private Dao dao;
7.    public ServiceImpl() {
8.    }
9.    public ServiceImpl(Dao dao) {
10.             this.dao=dao;
11.         }
12.         public void setDao(Dao dao) {
```

```
13.                 this.dao = dao;
14.         }
15.
16.         public void serviceMethod() {
17.                 System.out.println("This is Service Layer");
18.                 dao.daoMethod();
19.         }
20.   }
```

Servlet.java

```
1. package com.neo.spring.servlet;
2.
3. public interface Servlet {
4.       public void servletMethod();
5. }
```

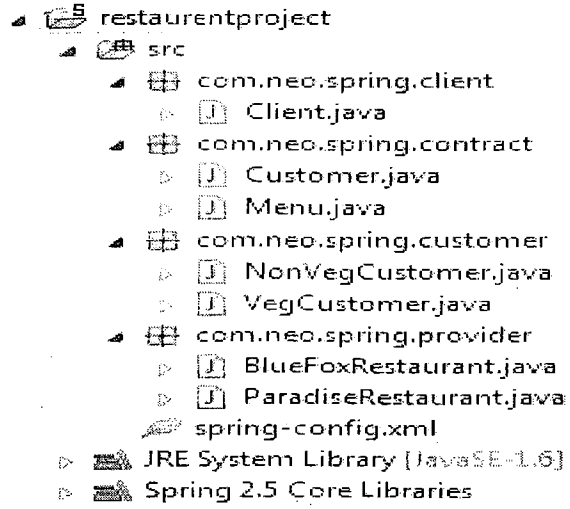
ServletImpl.java

```
1. package com.neo.spring.servlet;
2.
3. import com.neo.spring.service.Service;
4.
5. public class ServletImpl implements Servlet {
6.     private Service service;
7.     public ServletImpl() {
8.     }
9.     public ServletImpl(Service service) {
10.                 this.service=service;
11.         }
12.         public void setService(Service service) {
13.                 this.service = service;
14.         }
15.         public void servletMethod() {
16.                 System.out.println("This is presentaion layer");
17.                 service.serviceMethod();
18.         }
19.   }
```

applicationContext.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3.     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.     xmlns:p="http://www.springframework.org/schema/p"
5.     xsi:schemaLocation="http://www.springframework.org/schema/beans
6.     http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
7.
8.     <bean id="daoref" class="com.neo.spring.dao.DaoImpl">
9.     </bean>
10.         <bean id="serviceref" class="com.neo.spring.service.ServiceImpl">
11.             <property name="dao" ref="daoref"></property>
12.         </bean>
13.         <bean id="servletref" class="com.neo.spring.servlet.ServletImpl">
14.             <property name="service" ref="serviceref"></property>
15.         </bean>
16.
17.   </beans>
```

Restaurent IOC Project



Client.java:

```
1. package com.neo.spring.client;
2. import org.springframework.beans.factory.BeanFactory;
3. import org.springframework.beans.factory.xml.XmlBeanFactory;
4. import org.springframework.core.io.ClassPathResource;
5. import com.neo.spring.contract.Customer;
6. public class Client {
7.     private static BeanFactory factory = new XmlBeanFactory(
8.                 new ClassPathResource("spring-config.xml"));
9.
10.    public static void main(String[] args) {
11.                 Customer customer =(Customer)factory.getBean("customer");
12.                 customer.getFoodService();
13.         }
14.    }
```

Menu.java

```
1. package com.neo.spring.contract;
2.
3. public interface Menu {
4.     public abstract void getVegSerivice();
5.
6.     public abstract void getNonVegService();
7. }
```

Customer.java

```
1. package com.neo.spring.contract;
2.
3. public interface Customer {
4.     public abstract void getFoodService();
5. }
```

VegCustomer.java

```
1. package com.neo.spring.customer;
2.
3. import com.neo.spring.contract.Customer;
```

```
4. import com.neo.spring.contract.Menu;
5.
6. public class VegCustomer implements Customer {
7.
8.     private Menu menu;
9.
10.         public void setMenu(Menu menu) {
11.               this.menu = menu;
12.         }
13.
14.         @Override
15.         public void getFoodService() {
16.
17.               menu.getVegSerivice();
18.
19.         }
20.   }
```

NonVegCustomer.java

```
1. package com.neo.spring.customer;
2.
3. import com.neo.spring.contract.Customer;
4. import com.neo.spring.contract.Menu;
5.
6. public class NonVegCustomer implements Customer {
7.     private Menu menu;
8.
9.     public void setMenu(Menu menu) {
10.               this.menu = menu;
11.         }
12.
13.         @Override
14.         public void getFoodService() {
15.
16.               menu.getNonVegService();
17.         }
18.   }
```

BlueFoxRestaurant.java

```
1. package com.neo.spring.provider;
2.
3. import com.neo.spring.contract.Menu;
4.
5. public class BlueFoxRestaurant implements Menu {
6.
7.     @Override
8.     public void getVegSerivice() {
9.           System.out.println("BlueFoxRestaurant best veg service");
10.
11.         }
12.
13.         @Override
14.         public void getNonVegService() {
15.         System.out.println("BlueFoxRestaurant worst non-veg service");
16.
```

```
17.         }
18.   }
```

ParadiseRestaurant.java

```
1. package com.neo.spring.provider;
2.
3. import com.neo.spring.contract.Menu;
4.
5. public class ParadiseRestaurant implements Menu {
6.     @Override
7.     public void getVegSerivice() {
8.
9.           System.out.println("ParadiseRestaurant worst veg service");
10.
11.          }
12.
13.          @Override
14.          public void getNonVegService() {
15.
16.          System.out.println("ParadiseRestaurant best non-veg service");
17.
18.          }
19.   }
```

spring-config.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3.     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.     xmlns:p="http://www.springframework.org/schema/p"
5.     xsi:schemaLocation="http://www.springframework.org/schema/beans
6.     http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
7.
8.     <!--  provider configuration -->
9.     <bean id="paradiseref"
10.          class="com.neo.spring.provider.ParadiseRestaurant"></bean>
11.    <bean id="bluefoxref"
12.          class="com.neo.spring.provider.BlueFoxRestaurant"></bean>
13.
14.          <!-- customer configuration -->
15.    <!--
16.          <bean id="customer" class="com.neo.spring.customer.VegCustomer">
17.                <property name="menu">
18.                      <ref bean="bluefoxref" />
19.                </property>
20.          </bean>
21.    -->
22.       <bean id="customer" class="com.neo.spring.customer.NonVegCustomer">
23.                <property name="menu">
24.                      <ref bean="paradiseref" />
25.                </property>
26.       </bean>
27.    </beans>
```

IOC terminology

```
class student{ ----> Dependent
   int sno; ----> Dependency
   String sname;
   Address address; ----> Dependency
   // setters & getters

}
```

```
Class Address{ ----> Dependent
    int hno; ----> Dependency
    String street;
    // setters & getters

}
```

- Here the dependencies can be primitive datatypes or primitive object types or userdefined datatypes.

Q) How many ways we can give a value for dependencies?

In two ways:-

- Constructor Approach
- Setter Approach

Constructor Approach:

Address.java

```
public class Address {

}
```

Student.java

```
public class Student {
	private int sno;
	private String sname;
	private Address address;
	public Student(int sno, String sname, Address address) {
		this.sno=sno;
		this.sname=sname;
		this.address=address;
	}
}
```

Using constructor give the values to the dependencies.

```
Address add=new Address();
Student student=new Student(1001,'sekhar',add);
```

Setter Approach:

Address.java

```
public class Address {

}
```

Student.java

```
public class Student {
	private int sno;
	private String sname;
	private Address address;
	public Student(int sno, String sname, Address address) {
		this.sno=sno;
		this.sname=sname;
```

```
            this.address=address;
      }
      public Student(){

      }
      public int getSno() {
            return sno;
      }
      public void setSno(int sno) {
            this.sno = sno;
      }
      public String getSname() {
            return sname;
      }
      public void setSname(String sname) {
            this.sname = sname;
      }
      public Address getAddress() {
            return address;
      }
      public void setAddress(Address address) {
            this.address = address;
      }
}
```

Using setter methods giving the values to the dependencies.

```
Address address=new Address();
Student student=new Student();
student.setSno(1001);
student.sname('sekhar');
student.setAddress(address);
```

- Here we are writing default constructor to create the student object without passing any constructor arguments (to go for setter approach).
- Here we are writing the setter methods to give the values to dependencies .
- Here we are writing getter methods to access the values of dependencies. For setter approach actually getter methods are not required.

Q) What is IOC?

Ans: ***Inversion Of Control*** is a design principle using which an external entity injecting dependencies into dependent objects.

In Spring, Spring Container is the external entity, which injects dependency objects into dependent object.

Q) What is DI?

Ans:

- Physical realization of IOC is nothing but DI (Dependency Injection).
- Resolving the dependencies of among the beans is nothing but Dependency Injection.
- In case of spring , dependencies are injected into dependent object (beans) through DI by spring container.
  In spring DI is generally achieved in 2 ways,
  1) setter injection .
  2) constructor injection.
- In a well designed spring bean, interfaces are specified as dependencies.
- If dependency is provided to dependent object via setter method , it is known as setter injection.
- If dependency is provided to dependent object via constructor, it is known as constructor injection.

Q.) What are the Benfits of Dependency Injection?

Ans:

1.) Dynamically we can change dependency objects.
2.) Even dependency changing dependent object code remains same, there by it is offering loose coupling among dependent and dependencies.

Q.) What is Spring Bean?
Ans: Java class that are controlled by spring container are known as Spring Bean.

Q.) Which injection is better?
Ans: Setter injection is always preferable because we can change the dependency value any no. of times as we required.
But with constructor we can't change dependency object value, as and when we require.

Steps involved in Spring Application Development:

- Develop the Spring beans(classes) according to the Bussiness requirement.
- Develop the spring configuration file and configure the spring beans developed in step1.
- Inject the dependencies into dependent objects either in setter approach (identify how the bean is expecting Dependency object values based on that we are going to define either of one).
- Create the instance of spring container.
- Get the instance of dependent objects from the container by making a method call getBean(), by passing id of a bean.
- After we get the bean object we will call the required methods on dependent bean.

Q) How to implement setter injection in a spring container?
Ans:
Step 1: In spring bean class declare the dependency variable.
Step 2: Define setter method for that variable.
Step 3: Use <property > tag in bean configuration file & supply the value.

Example for Setter injection where dependency is primitive type:

```
public class  Purse{
        private double money;
        public void setMoney(double money){
                this.money=money;
         }
}
```

Spring configuration file:

```
<beans ...>
  <bean  id="purse" class="Purse">
    <property name="money" value="10"></property>
                        (or)
    <property name="money" value="10"/>
                        (or)
    <property name="money">
            < value>10</value>
     </property>
  </bean>
</beans>
```

Example for Setter injection where dependency is object type:

```
public class Provider {
}
```

```
public class Customer{
        private Provider provider;
        public void setProvider(Provider provider){
                this.provider= provider;
         }
}
```

Spring configuration file:

```
<beans ...>
  <bean id="p" class="Provider"/>
  <bean id="c" class="Customer">
    <property name="provider" ref="p"></property>
                    (or)
    <property name="provider" ref="p"/>
                    (or)
     <property name="provider">
             <ref bean="p"/>
      </property>
  </bean>
</beans>
```

Q) How to implement constructor injection in a spring container?

Ans:

Step 1: In spring bean class declare dependency variable.
Step 2: Define constructor which will take that variable as parameter.
Step 3: Use <constructor-arg> tag in bean configuration file & supply the value.

Example for constructor injection where dependency is primitive type:

```
public class  Purse{
        private double money;
        public Purse(double money){
                this.money=money;
         }
}
```

Spring configuration file:

```
<beans ...>
  <bean  id="purse" class="Purse">
    <constructor-arg value="10"></ constructor-arg>
                    (or)
    <constructor-arg value="10"/>
                    (or)
    < constructor-arg>
             < value>10</value>
     </constructor-arg>
  </bean>
</beans>
```

Example for Constructor injection where dependency is object type:

```
public class Provider {
}

public class Customer{
        private Provider provider;
        public Customer(Provider provider){
```

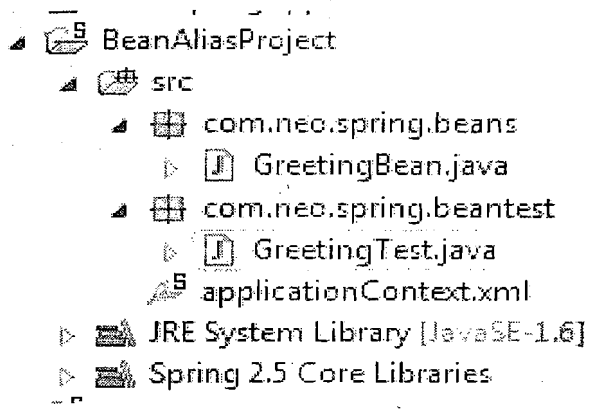
```
                this.provider= provider;
            }
        }
```

Spring configuration file:

```
<beans ...>
  <bean id="providerRef" class="Provider"/>
  <bean id="cuctomerRef" class="Customer">
    <constructor-arg ref="providerRef></constructor-arg>
                    (or)
    <constructor-arg ref="providerRef/>
                    (or)
     <constructor-arg>
            <ref bean="providerRef"/>
      </constructor-arg>
  </bean>
</beans>
```

BeanAliasProject

Bean alias name can be used to define multiple names for a configured spring bean. We can define alias names with "name" attribute of <bean> tag.



GreetingBean.java

```
1. package com.neo.spring.beans;
2. public class GreetingBean{
3.     private String name;
4.
5.     public void setName(String name) {
6.           this.name=name;
7.     }
8.
9.     public String getName() {
10.                return name;
11.          }
12.    }
```

applicationContext.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans
3.     xmlns="http://www.springframework.org/schema/beans"
4.     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5.     xmlns:p="http://www.springframework.org/schema/p"
6.     xsi:schemaLocation="http://www.springframework.org/schema/beans
7.     http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
8.
9. <!- - here we can write bean alias using name attribute - - >
10.          <bean id="gb" name="gb1,gb2,gb3,gb4"
11.                      class="com.neo.spring.beans.GreetingBean">
12.                <property name="name" value="welcome to NIT" ></property>
13.          </bean>
14.    </beans>
```

GreetingTest.java

```
1. package com.neo.spring.beantest;
2. import org.springframework.context.ApplicationContext;
3. import org.springframework.context.support.ClassPathXmlApplicationContext;
```

```
4. import com.neo.spring.beans.GreetingBean;
5. public class GreetingTest {
6.
7. private static ApplicationContext context = new
8.                 ClassPathXmlApplicationContext("applicationContext.xml");
9.     public static void main(String[] args) {
10.
11.           GreetingBean greetingBean = (GreetingBean)context.getBean("gb");
12.           String name=greetingBean.getName();
13.           System.out.println(name);
14.           String aliasNames[] = context.getAliases("gb");
15.           System.out.println("The following are Alias names of gb:");
16.           for (int i = 0; i < aliasNames.length; i++) {
17.                System.out.println(aliasNames[i]);
18.       }
19.      }
20.   }
```

Output

```
welcome to NIT
The following are Alias names of gb:
gb2
gb1
gb3
gb4
```

Note:

```
<bean name="gb1,gb2,gb3,gb4"
          class="com.neo.spring.beans.GreetingBean">
```

we can also write like above in configuration file without ***id*** attribute,here the container creates object based on given alias name.

ex:

```
GreetingBean greetingBean = (GreetingBean) context.getBean("gb1");
```

Here the container treats *'gb1'* as unique ***id*** reference, remaining references as alias to that unique id

If we print alias names now then the output is:

```
String aliasNames[] = context.getAliases("gb1");
       System.out.println("Now the Alias names will be as follows:");
       for (int i = 0; i < aliasNames.length; i++) {
            System.out.println(aliasNames[i]);
```

Output

```
welcome to NIT
Now the Alias names will be as follows:
gb2
gb3
gb4
```

So here the 'gb1' treated as unique id references, remaining as aliases

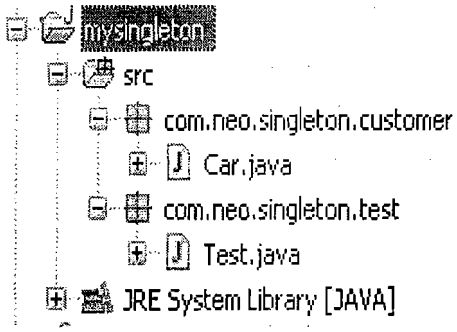
Key notes on **id, name** attributes :

- ***Id*** attribute is used to define reference of configured spring bean.
- ***id*** value should be ***unique,*** *whereas* ***name*** *no need to be* ***uniqure.***
- ***Id*** value won't allow special characters (like /, _,*...etc). whereas ***name*** attribute allows any kind of special characters.
- The ***name*** attribute is useful when we develop spring MVC based web applications, because As we know every URL should start with **"/".**
- Generally most of the cases we use ***id*** attribute only but not ***name*** attribute.

Singleton

Generally we create singleton when there is no instance variables and static variables to save the memory.

Mysingleton



Car.java

```
1. package com.neo.singleton.customer;
2. public class Car {
3.     private static Car car=new Car();
4.     private Car(){
5.           System.out.println("Car() constructor");
6.     }
7.     public static Car getInstance(){
8.           return car;
9.     }
10.    }
```

Test.java

```
1. package com.neo.singleton.test;
2. import com.neo.singleton.customer.Car;
3. public class Test {
4.     public static void main(String[] args) {
5.           Car car1=Car.getInstance();
6.           Car car2=Car.getInstance();
7.           Car car3=Car.getInstance();
8.     Car car4=Car.getInstance();
9.     Car car5=Car.getInstance();
10.          Car car6=Car.getInstance();
11.          System.out.println(car1);
12.          System.out.println(car2);
13.          System.out.println(car3);
14.          System.out.println(car4);
15.          System.out.println(car5);
16.          System.out.println(car6);
```

```
17.
18.    }
19.    }
```

output

```
Car() constructor
com.neo.singleton.customer.Car@360be0
com.neo.singleton.customer.Car@360be0
com.neo.singleton.customer.Car@360be0
com.neo.singleton.customer.Car@360be0
com.neo.singleton.customer.Car@360be0
com.neo.singleton.customer.Car@360be0
```

- So this is the way to create singleton object. But here we are implementing our own logic to implement singleton.
- In the case of spring singleton design pattern is implicitly implemented. To make one object as singleton, or non-singleton just we need change confiuration in the spring configuration file. But we no need to implement our own logic.

spring scopes

This is just an empty bean our aim is to find out how the container is creating objects when we try to call this bean. Let's see this by the examples.

SpringBean.java

```
1. package com.neo.spring.bean;
2. public class SpringBean {
3.
4. }
```

applicationContext.xml(configuration file)

without any attribute declarations in the <bean> tag.

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3.     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.     xsi:schemaLocation="http://www.springframework.org/schema/beans
5. http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">
6.     <bean id="sb"
7.           class="com.neo.spring.bean.SpringBean">
8.     </bean>
9. </beans>
```

Here we are testing the application.

Client.java

```
1. package com.neo.spring.client;
2. import org.springframework.context.ApplicationContext;
3. import org.springframework.context.support.ClassPathXmlApplicationContext;
4. import com.neo.spring.bean.SpringBean;
5. public class Client{
6.    private static ApplicationContext context = new
7.       ClassPathXmlApplicationContext(
8.          "/com/neo/somashekar/spring/config/applicationContext.xml");
9.
10.         public static void main(String[] args) {
11.               SpringBean springBean1 = (SpringBean)
12.                                                   context.getBean("sb");
13.               System.out.println("object 1-->" + springBean1);
14.               SpringBean springBean2= (SpringBean) context.getBean("sb");
15.               System.out.println("object 2-->" + springBean2);
16.               SpringBean springBean 3= (SpringBean)
17.                                                   context.getBean("sb");
18.               System.out.println("object 3-->" + springBean3);
19.         }
20.   }
```

Output

```
object 1--> com.neo.spring.bean.SpringBean @1d7fbfb
object 2--> com.neo.spring.bean.SpringBean@1d7fbfb
object 3--> com.neo.spring.bean.SpringBean @1d7fbfb
```

Analysis

By observing the above application the spring container creates only one object (returning same address) even we call getBean("objRef") 'n' no. of times. So by default a spring bean is "SingleTon".

- If we want to change its nature to "Non-SingleTon" i.e. for each 'getBean ()'method call on container one brand new instance has to be created. To achieve this we can make use of "**scope**" attribute of **<bean>** tag.
- Scope attribute takes two values:
  1) **singleton**
  2) **prototype**
- ⇨ Let's see this by our application by rewriting configuration file i.e. applicationContext.xml file

applicationContext.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3.    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.    xsi:schemaLocation="http://www.springframework.org/schema/beans
5.    http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">
6.    <bean id="sb" class=" com.neo.spring.bean.SpringBean "
7.          scope="singleton">
8.    </bean>
9. </beans>
```

Now run the Client.java again, then we will get output something like this shown below

Output

```
object 1--> com.neo.spring.bean.SpringBean@1d7fbfb
object 2--> com.neo.spring.bean.SpringBean@1d7fbfb
object 3--> com.neo.spring.bean.SpringBean@1d7fbfb
```

Analysis

Even with or without declaring the scope attribute we are getting the same output. So by default the spring bean is singleton.
Now if we want the same spring bean non-singleton we can make use of scope attribute value i.e. prototype.

- ⇨ Let's rewrite the application and observe the output by using the scope attribute value in configuration file.

applicationContext.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3.     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.     xsi:schemaLocation="http://www.springframework.org/schema/beans
5.     http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">
6.     <bean id="sb" class="com.neo.spring.bean.SpringBean"
7.           scope="prototype" >
8.     </bean>
9. </beans>
```

Now run the Client.java again, then we will get output something like this shown below

output

```
object 1--> com.neo.spring.bean.SpringBean@1e57e8f
object 2--> com.neo.spring.bean.SpringBean@1d7fbfb
object 3--> com.neo.spring.bean.SpringBean@e020c9
```

Analysis

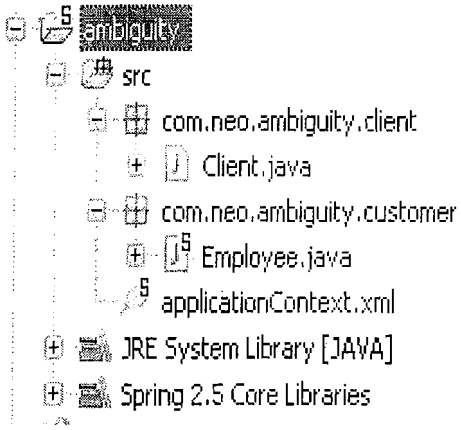
In the above output the spring container has created three different objects i.e. we made spring bean as Non-SingleTon by making use of 'scope=prototype'.

Note: In old versions of spring we used to use "**singleton**" attribute of **<bean>** tag explicitly to achieve single ton and non-singleton nature.

All possible values of scope attribute

a. **singleton**
⇨ Specifies to container to create the configured spring bean only once in the container.
b. **prototype**
⇨ Specifies the container to create the configured spring bean for each time when it is requested (getBean("springref")).
c. **request**
⇨ Specifies the container to create the configured spring bean once in per web request. It is applicable only in web module.
d. **session**
⇨ Specifies the container to create the configured spring bean once per HttpSession. It is applicable only web module.
e. **globalsession**
⇨ specifies the container to crate the configured spring bean once in a global session. It is applicable in portals.
f. **thread**
⇨ Specifies the container to create the configured spring bean once per each thread. It is not implicitly registered. If we want to specify thread as a scope for bean, we have to register explicitly.

Constructor Ambiguity



Employee.java

```
1. package com.neo.ambiguity.customer;
2.
3. public class Employee {
4.    private int eno=1;
5.    private String name=null;
6.    private double sal=0.0;
7.    private String desig="clerk";
8.    public Employee(int eno,String name) {
9.          this.eno=eno;
10.               this.name=name;
11.         }
12.         public Employee(double sal,String desig) {
13.               this.sal=sal;
14.               this.desig=desig;
15.         }
16.
17.         public void getEmployeeDetails(){
18.               System.out.println(eno);
19.               System.out.println(name);
20.               System.out.println(sal);
21.               System.out.println(desig);
22.         }
23.   }
```

Client.java

```
1. package com.neo.ambiguity.client;
2.
3. import org.springframework.context.ApplicationContext;
4. import org.springframework.context.support.ClassPathXmlApplicationContext;
5. import com.neo.ambiguity.customer.Employee;
6.
7. public class Client {
8.    private static ApplicationContext context=new
9.                   ClassPathXmlApplicationContext("applicationContext.xml");
10.         public static void main(String[] args) {
11.               Employee employee =(Employee)context.getBean("empref");
12.               employee.getEmployeeDetails();
13.         }
14.   }
```

applicationContext.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans
3.     xmlns="http://www.springframework.org/schema/beans"
4.     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5.     xsi:schemaLocation="http://www.springframework.org/schema/beans
6.     http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
7.
8.     <bean id="empref" class="com.neo.ambiguity.customer.Employee">
9.           <constructor-arg value="1001" ></constructor-arg>
10.          <constructor-arg value="sekhar" ></constructor-arg>
11.    </bean>
12.    </beans>
```

In the above program, we can't expect the output. Because Employee class having two constructors with two arguments and it takes the value in String and it converts into corresponding type. So it may assign to Employee(eno,ename) or Employee(sal,desig).

output

```
1
null
1001.0
sekhar
```

Here, it assign to Employee(sal,desig). To avoid this ambiguity problem,
It has two attributes.

1.) **index** → it specifies the index of the argument (it takes from '0').
2.) **type** → it specifies the type of the argument. For object type we have to give fully qualified name. For primitives directly we can give the corresponding privitive key words.

Now modify "applicationContext.xml" to avoid the ambiguity problem as follows.

applicationContext.xml

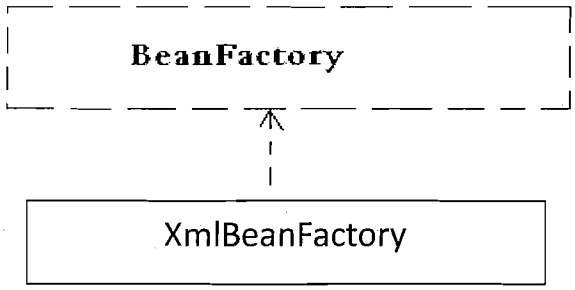
```
1. <beans
2.     xmlns="http://www.springframework.org/schema/beans"
3.     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.     xsi:schemaLocation="http://www.springframework.org/schema/beans
5.     http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
6.      <bean id="empref" class="com.neo.ambiguity.customer.Employee">
7.        <constructor-arg value="1001" index="0" type="int" /.
8.        <constructor-arg value="sekhar" index="1" type="java.lang.String" />
9.      </bean>
10.   </beans>
```

output

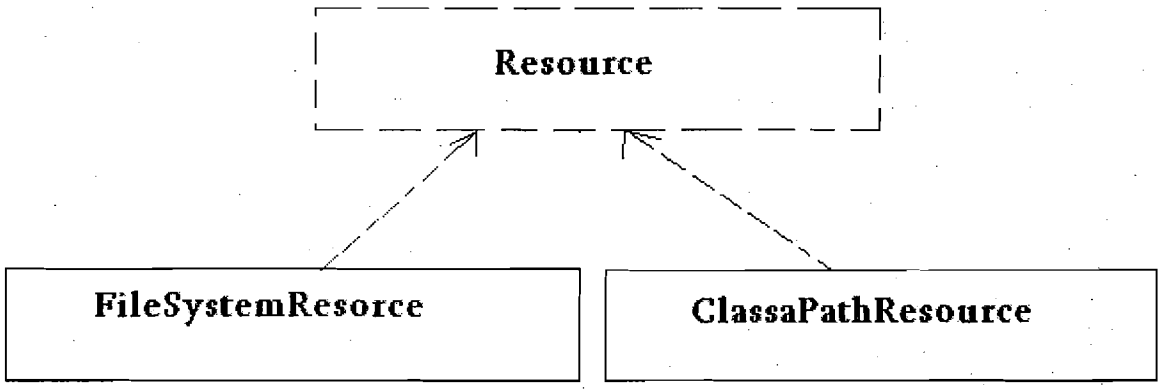
```
1001
sekhar
0.0
clerk
```

Q) Describe the different types of containers in spring framework?
Ans: Spring containers are of 2 types.
1) Bean Factory based.
2) ApplicationContext based.

Bean Factory based
BeanFactory is an interface .it's implementation class is XmlBeanFactory.



- ⇨ Create instance of **XmlBeanFactory** class , is nothing but we create spring container.
- ⇨ Constructor of **XmlBeanFactory** class takes **Resource** object as argument, which represents spring configuration file.
- ⇨ There are two implementation classes of **Resource** interface, which are used to specify spring configuration file.



Using XmlBeanFactory with FileSystemResource container

```
1. package com.neo.spring.client;
2. import org.springframework.beans.factory.BeanFactory;
3. import org.springframework.beans.factory.xml.XmlBeanFactory;
4. import org.springframework.core.io.FileSystemResource;
5. public class Client {
6.    private static BeanFactory beanFactory = new XmlBeanFactory(
7.          new FileSystemResource("E:/SomasekharReddy/Struts/workspace
8.                            /springcontainer/src/applicationContext.xml"));
9.
10.         public static void main(String[] args) {
11.               System.out.println(beanFactory.getBean("springBeanRef"));
12.
13.         }
14.   }
```

In the above example FileSystemResource is taking complete system path(absolute path). So in the future if we change location of the project then again we need to change the location of spring configuration file in the program. That's why it is not advisable to use FileSystemResource.

Using XmlBeanFactory with ClassPathResource container

```
1. package com.neo.spring.client;
2. import org.springframework.beans.factory.BeanFactory;
3. import org.springframework.beans.factory.xml.XmlBeanFactory;
4. import org.springframework.core.io.ClassPathResource;
5.
6. public class Client {
7.     private static BeanFactory beanFactory = new XmlBeanFactory(new
8.                                 ClassPathResource("applicationContext.xml"));
9.     public static void main(String[] args) {
10.                 System.out.println(beanFactory.getBean("springBeanRef"));
11.
12.         }
13.     }
```

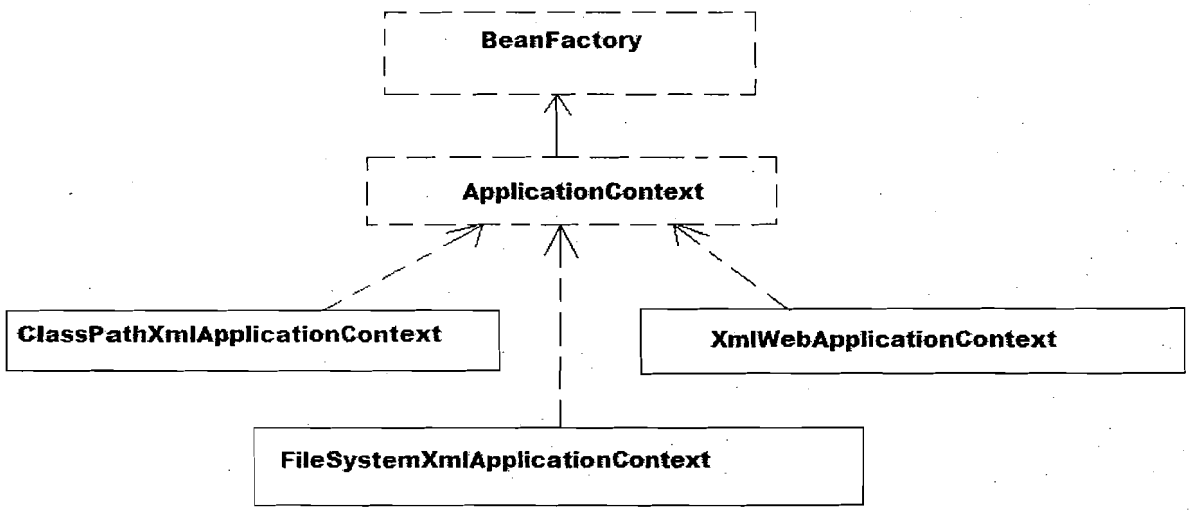
In the above example container reading the configuration file from class path, but we are not specifying the complete path. Even we change the location of the project in the system, still our application work without modification. Because it reads configuration file from classpath. So it is advisable to use ClassPathResource instead of FileSystemResource.

Key notes on BeanFactory based containers

- ⇨ It is a light weight container.
- ⇨ It loads spring beans configured in spring configuration file, and ménages the life cycle of the spring bean when we call getBean("springbeanref"). So when we call getBean("springbeanref") then only spring bean life cycle starts.
- ⇨ Generally it is used to develop stand alone applications, mobile applications.

ApplicationContext based

It is a child interface of BeanFactory.



Using FileSystemXmlApplicationContext

```
1. package com.neo.spring.client;
2.
```

```
3. import org.springframework.context.ApplicationContext;
4. import org.springframework.context.support.FileSystemXmlApplicationContext;
5.
6. public class Client {
7.    private static ApplicationContext context = new FileSystemXmlApplicationContext(
8.   "E:/SomasekharReddy/Struts/workspace/springcontainer/src/applicationContext.xml");
9.
10.         public static void main(String[] args) {
11.               System.out.println(context.getBean("springBeanRef"));
12.
13.         }
14.   }
```

If we use FileSystemXmlApplicationContext container, we need to pass absolute path of the spring configuration file to this constructor. So, if the project location is changes we need to change the our application. So it is not advisable to use this container.

Using ClassPathXmlApplicationContext

```
1. package com.neo.spring.client;
2.
3. import org.springframework.context.ApplicationContext;
4. import org.springframework.context.support.ClassPathXmlApplicationContext;
5.
6. public class Client {
7.    private static ApplicationContext context = new
8. ClassPathXmlApplicationContext("applicationContext.xml");
9.    public static void main(String[] args) {
10.               System.out.println(context.getBean("springBeanRef"));
11.
12.         }
13.   }
14.
```

In the above example container reading the configuration file from class path, but we are not specifying the complete path (absolute path). Even we change the location of the project in the system, still our application work without modification. Because it reads configuration file from classpath. So it is advisable to use ClassPathXmlApplicationContext container.

Using XmlWebApplicationContext

This is used in sprinb webmvc module. This container object created by web container internally. We no need to create on our own.

Key notes on ApplicationContext

- It loads spring beans configured in spring configuration file, and ménages the life cycle of the spring bean as and when container starts, it won't wait until getBean() method is called.
- ApplicationContext based container provides all the features of BeanFactory. In addition to them it provides the following features also.
    1) Event-handling support
    2) Internationalization support
    3) Remoting
    4) EJB Intigration
    5) Scheduling
    6) JNDI look-up ...etc.

In enterprise level application always ApplicationContext based container is used.

Spring bean life-cycle

Spring container controls the life cycle of spring bean i.e. from instantiation to destruction.

Spring bean has 4 life cycle states:

- Instantiation
- Initialization
- Ready to use (method ready to use state)
- Destruction

⇨ initialization method, destruction method can be given to spring been in the following ways

- By implementing spring framework given interfaces they are
  1. InitializingBean
  2. DisposableBean
- By configuring custom initialization and destruction method in the spring configuring file using <bean> tag attributes
  - init-method
  - destroy-method
- By using annotations
  - PreDestroy
  - PostConstruct

Note: if spring bean is implementing framework interface "InitializingBean", we need to define afterPropertiesSet() in bean class. It becomes the initialization method .Similarly if bean implements "DesposableBean" interface , destroy method becomes destruction method & no need to inform.

Q) What is BeanNameAware and why it is used?

Ans: BeanNameAware is an interface and it has one method i.e., **setBeanName(String)**, by overriding this method we will get the name(configured name) of the bean. This is used to know the configuration name of the spring bean.

Ex:
```
public class SpringBean implements BeanNameAware{
      private String name;
      public void setName(String name) {
            this.name = name;
      }
      public void setBeanName(String beanName) {
            System.out.println("beanName is:"+beanName);
      }
}
```

Q) What is BeanFactoryAware?

Ans: BeanFactoryAware is an interface. It is used to get the container information in which spring bean is available.

It has one method i.e., **setBeanFactory(BeanFactory)**.

Ex:
```
public class SpringBean implements BeanFactoryAware {
      private String name;
      public void setName(String name) {
            this.name = name;
      }
      public void setBeanFactory(BeanFactory factory) throws BeansException {
            System.out.println(factory);
      }
}
```

Q) What is ApplicationContextAware?

Ans: ApplicationContextAware is an interface, it is also contain one method i.e., **setApplicationContext(ApplicationContext).** It is also used to get the container information in which spring bean is available.

Ex:
```
public class SpringBean implements ApplicationContextAware{
      private String name;
      public void setName(String name) {
            this.name = name;
      }
      public void setApplicationContext(ApplicationContext context)
                  throws BeansException {
      System.out.println(context);
      }
}
```

BeanPostProcessor:

BeanPostProcessor is an interface.
It has two methods:

- postProcessBeforeIntialization(Object, String)
- postProcessAfterInitialization(Object, String)

BeanPostPocessor is used to provide some common logic to all beans or to set of beans which are configured in spring configuration file.

- postProcessBeforeIntialization() method is executed before the initialization method called.
- postProcessAfterInitialization() method is executed after the intialization method called.

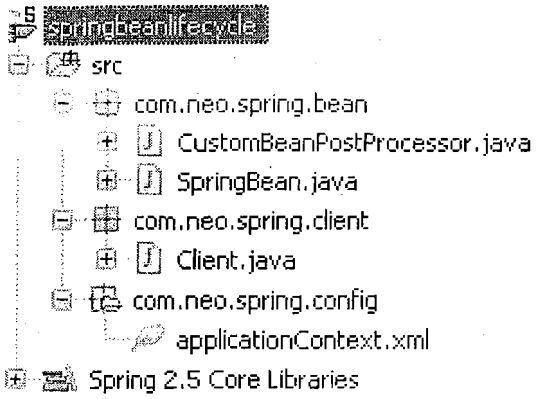
These methods will be called for each spring bean, which is configured in spring configuration file.

Q) How to use BeanPostProcessor?

Ans: We have to write some class and we have to implement **BeanPostProcessor** interface. And we have to override the postProcessBeforeIntialization() and postProcessAfterInitialization() methods.

Then register user defined **BeanPostProcessor** object with spring container. i.e configure the user defined BeanPostProcessor bean in spring configuration file.

SpringBean Life Cycle



SpringBean.java

```
1. package com.neo.spring.bean;
2. import javax.annotation.PostConstruct;
3. import javax.annotation.PreDestroy;
4. import org.springframework.beans.BeansException;
5. import org.springframework.beans.factory.BeanFactory;
6. import org.springframework.beans.factory.BeanFactoryAware;
7. import org.springframework.beans.factory.BeanNameAware;
8. import org.springframework.beans.factory.DisposableBean;
9. import org.springframework.beans.factory.InitializingBean;
10.   import org.springframework.context.ApplicationContext;
11.   import org.springframework.context.ApplicationContextAware;
12.
13.   public class SpringBean implements InitializingBean, DisposableBean,
14.               BeanNameAware, BeanFactoryAware, ApplicationContextAware {
15.         private String name;
16.
17.         public SpringBean() {
18.               sop("SpringBean() ");
19.         }
20.
21.         public void setName(String name) {
22.               this.name = name;
23.               sop("SpringBean.setName(String name) ");
24.         }
25.
26.         public String getName() {
27.               // Service method of SpringBean
28.               sop("SpringBean is servicing now. this state is called
29.                                                   method ready state ");
30.               return name;
31.         }
32.
33.         @Override
34.         public void afterPropertiesSet() throws Exception {
35.               sop("InitializingBean.afterPropertiesSet()");
36.
37.         }
38.
39.         @Override
```

```
40.         public void destroy() throws Exception {
41.               sop("DisposableBean.destroy()");
42.
43.         }
44.
45.         public void configuratinInit() {
46.               sop("init-method.configuratinInit()");
47.         }
48.
49.         public void configurationDestroy() {
50.               sop("destroy-method.configurationDestroy()");
51.         }
52.
53.         @PostConstruct
54.         public void annotationInit() {
55.               sop("PostConstruct.annotationInit()");
56.         }
57.
58.         @PreDestroy
59.         public void annotationDestroy() {
60.               sop("PreDestroy.annotationDestroy()");
61.         }
62.
63.         // knowing spring bean name
64.         @Override
65.         public void setBeanName(String beanName) {
66.               sop("BeanNameAware.setBeanName(String beanName) : " +
67.                                                                  beanName);
68.
69.         }
70.
71.         // where spring bean is residing
72.         @Override
73.         public void setBeanFactory(BeanFactory factory) throws
74.                                                          BeansException {
75.               sop("BeanFactoryAware.setBeanFactory(BeanFactory factory) :
76.                                                            " + factory);
77.
78.         }
79.
80.         // where spring bean is residing
81.         @Override
82.         public void setApplicationContext(ApplicationContext context)
83.                     throws BeansException {
84.
85.     sop("ApplicationContextAware.setApplicationContext(ApplicationContext
86.                                              context) : "  + context);
87.
88.         }
89.
90.         public void sop(Object object) {
91.               System.out.println(object);
92.         }
93.   }
```

CustomBeanPostProcessor.java

```
1. package com.neo.spring.bean;
2.
3. import org.springframework.beans.BeansException;
4. import org.springframework.beans.factory.config.BeanPostProcessor;
5. public class CustomBeanPostProcessor implements BeanPostProcessor {
6.
7.     @Override
8.     public Object postProcessBeforeInitialization(Object beanObj,
9.                 String beanName) throws BeansException {
10.
11.         System.out.println("CustomBeanPostProcessor.postProcessBefore
12.                         Initialization(Object beanObj, String beanName)");
13.                 System.out.println("BeanObject is : " + beanObj);
14.                 System.out.println("Bean Name is : " + beanName);
15.
16.                 return beanObj;
17.         }
18.
19.         @Override
20.         public Object postProcessAfterInitialization(Object beanObj,
21.                                     String beanName) throws BeansException {
22.
23.            System.out.println("CustomBeanPostProcessor.postProcessAfter
24.                         Initialization(Object beanObj, String beanName)");
25.                 System.out.println("BeanObject is : " + beanObj);
26.                 System.out.println("Bean Name is : " + beanName);
27.
28.                 return beanObj;
29.         }
30.   }
```

Client.java

```
1. package com.neo.spring.client;
2. import org.springframework.context.support.AbstractApplicationContext;
3. import org.springframework.context.support.ClassPathXmlApplicationContext;
4.
5. import com.neo.spring.bean.SpringBean;
6.
7. public class Client {
8.
9.    private static AbstractApplicationContext context = new
10.                       ClassPathXmlApplicationContext(
11.                      "com/neo/spring/config/applicationContext.xml");
12.
13.         public static void main(String[] args) {
14.
15.                 SpringBean bean = (SpringBean)context.getBean(
16.                                                         "springBeanRef");
17.                 String name = bean.getName();
18.                 System.out.println("Injected value is : "+name);
19.                 context.close();
20.         }
21.   }
```

applicationContext.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3.     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.     xmlns:context="http://www.springframework.org/schema/context"
5.     xsi:schemaLocation="http://www.springframework.org/schema/beans
6.         http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
7.         http://www.springframework.org/schema/context
8.      http://www.springframework.org/schema/context/spring-context-2.5.xsd">
9.
10.         <context:annotation-config />
11.         <bean name="springBeanRef" class="com.neo.spring.bean.SpringBean"
12.               init-method="configuratinInit" destroy-method="
13.                                                   configurationDestroy">
14.               <property name="name">
15.                     <value>lifecyclebean</value>
16.               </property>
17.         </bean>
18.
19.         <bean class="com.neo.spring.bean.CustomBeanPostProcessor"></bean>
20.
21.   </beans>
```

Output

```
SpringBean()
SpringBean.setName(String name)
BeanNameAware.setBeanName(String beanName) : springBeanRef
BeanFactoryAware.setBeanFactory(BeanFactory factory) :
org.springframework.beans.factory.support.DefaultListableBeanFactory@476128: defining
beans
[org.springframework.context.annotation.internalCommonAnnotationProcessor,org.springframe
work.context.annotation.internalAutowiredAnnotationProcessor,org.springframework.context.
annotation.internalRequiredAnnotationProcessor,springBeanRef,com.neo.spring.bean.CustomBe
anPostProcessor#0]; root of factory hierarchy
ApplicationContextAware.setApplicationContext(ApplicationContext context) :
org.springframework.context.support.ClassPathXmlApplicationContext@19a0c7c: display name
[org.springframework.context.support.ClassPathXmlApplicationContext@19a0c7c]; startup
date [Sun Feb 27 18:13:42 IST 2011]; root of context hierarchy
PostConstruct.annotationInit()
CustomBeanPostProcessor.postProcessBeforeInitialization(Object beanObj, String beanName)
BeanObject is :   com.neo.spring.bean.SpringBean@e61a35
Bean Name is : springBeanRef
InitializingBean.afterPropertiesSet()
init-method.configuratinInit()
CustomBeanPostProcessor.postProcessAfterInitialization(Object beanObj, String beanName)
BeanObject is :   com.neo.spring.bean.SpringBean@e61a35
Bean Name is : springBeanRef
PreDestroy.annotationDestroy()
DisposableBean.destroy()
destroy-method.configurationDestroy()
```

Q.) How to give same init, destroy method for all the spring beans configured in spring configuration file?

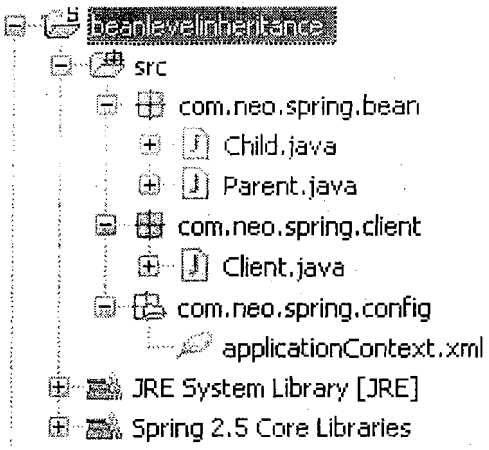
- ⇨ By configuring default-init, default-destroy attributes of <beans> tag , we can specify same init, destroy methods for all the spring beans configured in spring configuration file.
- ⇨ <bean xmlns="http....." **default-init-method="x" default-destroy-method=" y"/>

Inheritance

We have two types of inheritances.

- Bean level inheritance
  - Configuring child bean, with its own properties as well as with parent properties.
- Bean Configuration level inheritance
  - Reusing bean configuration in other bean configuration

Bean level inheritance



Child.java

```
1. package com.neo.spring.bean;
2. public class Child extends Parent {
3.     private String badHabbits;
4.     private String knowledge;
5.     public void setBadHabbits(String badHabbits) {
6.           this.badHabbits = badHabbits;
7.     }
8.     public void setKnowledge(String knowledge) {
9.           this.knowledge = knowledge;
10.          }
11.          public void displayDetails() {
12.                System.out.println(getMoney());
13.                System.out.println(getPropDocs());
14.                System.out.println(getDiseases());
15.                System.out.println(badHabbits);
16.                System.out.println(knowledge);
17.          }
18.    }
```

Parent.java

```
1. package com.neo.spring.bean;
2. public class Parent {
3.     private double money;
4.     private String propDocs;
5.     private String diseases;
6.     public void setMoney(double money) {
7.           this.money = money;
8.     }
9.     public void setPropDocs(String propDocs) {
10.                this.propDocs = propDocs;
11.          }
```

```
12.         public void setDiseases(String diseases) {
13.               this.diseases = diseases;
14.         }
15.         public double getMoney() {
16.               return money;
17.         }
18.         public String getPropDocs() {
19.               return propDocs;
20.         }
21.         public String getDiseases() {
22.               return diseases;
23.         }
24.   }
```

applicationContext.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans
3.     xmlns="http://www.springframework.org/schema/beans"
4.     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5.     xsi:schemaLocation="http://www.springframework.org/schema/beans
6.     http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
7.
8.     <bean id="child" class="com.neo.spring.bean.Child">
9.           <!—Here we are configuring parent properties in the
10.                Child class configuration -->
11.
12.               <property name="money" value="100"/>
13.               <property name="propDocs" value="debtdocs"/>
14.               <property name="diseases" value="nodiseases"/>
15.
16.               <property name="badHabbits"   value="smoking"/>
17.               <property name="knowledge" value="null"/>
18.          </bean>
19.   </beans>
```

Client.java

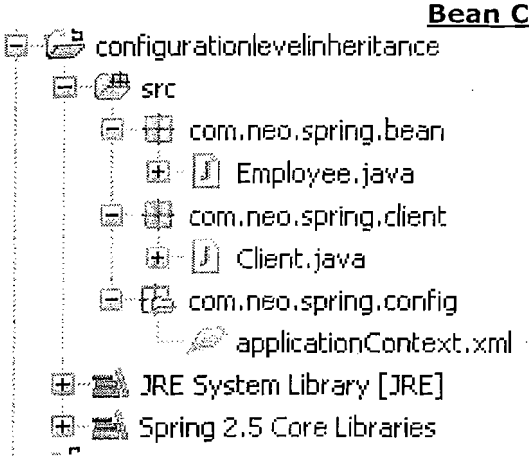
```
1. package com.neo.spring.client;
2. import org.springframework.context.support.AbstractApplicationContext;
3. import org.springframework.context.support.ClassPathXmlApplicationContext;
4. import com.neo.spring.bean.Child;
5. public class Client {
6.     private static AbstractApplicationContext context=new
7.                       ClassPathXmlApplicationContext("com/neo/spring/config/
8.                                                   applicationContext.xml");
9.     public static void main(String[] args) {
10.               Child child=(Child)context.getBean("child");
11.               child.displayDetails();
12.         }
13.   }
```

Output

```
100.0
debtdocs
nodiseases
```

```
smoking
null
```

Bean Configuration level inheritance



Employee.java

```
1. package com.neo.spring.bean;
2. public class Employee {
3.     private int eno;
4.     private String ename;
5.     private double sal;
6.     private String desig;
7.     private String hno;
8.     private String city;
9.           public void setEno(int eno) {
10.                this.eno = eno;
11.          }
12.          public void setEname(String ename) {
13.                this.ename = ename;
14.          }
15.          public void setSal(double sal) {
16.                this.sal = sal;
17.          }
18.          public void setDesig(String desig) {
19.                this.desig = desig;
20.          }
21.          public void setHno(String hno) {
22.                this.hno = hno;
23.          }
24.          public void setCity(String city) {
25.                this.city = city;
26.          }
27.          public void displayEmpDetails() {
28.                System.out.println(eno);
29.                System.out.println(ename);
30.                System.out.println(sal);
31.                System.out.println(desig);
32.                System.out.println(hno);
33.                System.out.println(city);
34.          }
35.    }
```

applicationContext.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans
3.     xmlns="http://www.springframework.org/schema/beans"
4.     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5.     xsi:schemaLocation="http://www.springframework.org/schema/beans
6.     http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
7.
8.     <bean id="emp1" class="com.neo.spring.bean.Employee">
9.           <property name="eno" value="1001"/>
10.                <property name="ename" value="sekhar"/>
11.                <property name="sal" value="2500"/>
12.                <property name="desig" value="teammember"/>
13.                <property name="hno" value="1-52-A"/>
14.                <property name="city" value="Gunipalli"/>
15.          </bean>
16.
17.          <bean id="emp2" class="com.neo.spring.bean.Employee">
18.                <property name="eno" value="1002"/>
19.                <property name="ename" value="somu"/>
20.                <property name="sal" value="2500"/>
21.                <property name="desig" value="teammember"/>
22.                <property name="hno" value="1-52-B"/>
23.                <property name="city" value="Gunipalli"/>
24.          </bean>
25.
26.          <bean id="emp3" class="com.neo.spring.bean.Employee">
27.                <property name="eno" value="1003"/>
28.                <property name="ename" value="somasekhar"/>
29.                <property name="sal" value="2500"/>
30.                <property name="desig" value="teammember"/>
31.                <property name="hno" value="1-52-B"/>
32.                <property name="city" value="Gunipalli"/>
33.          </bean>
34.    </beans>
```

Client.java

```
1. package com.neo.spring.client;
2. import org.springframework.context.support.AbstractApplicationContext;
3. import org.springframework.context.support.ClassPathXmlApplicationContext;
4. import com.neo.spring.bean.Employee;
5. public class Client {
6.     private static AbstractApplicationContext context=new
7.                              ClassPathXmlApplicationContext(
8.                              "com/neo/spring/config/applicationContext.xml");
9.     public static void main(String[] args) {
10.                Employee employee1=(Employee)context.getBean("emp1");
11.                employee1.displayEmpDetails();
12.                System.out.println();
13.
14.                Employee employee2=(Employee)context.getBean("emp2");
15.                employee2.displayEmpDetails();
16.                System.out.println();
17.
18.                Employee employee3=(Employee)context.getBean("emp3");
```

```
19.                 employee3.displayEmpDetails();
20.         }
21.   }
```

output

```
1001
sekhar
2500.0
teammember
1-52-A
Gunipalli

1002
somu
2500.0
teammember
1-52-B
Gunipalli

1003
somasekhar
2500.0
teammember
1-52-B
Gunipalli
```

Note: In the above bean configuration all the three employees having same city name, same salary, same designation. So why don't we reuse the bean configuration.

- ⇨ To reuse the bean configuration **<bean>** tag has **parent** attribute.
- ⇨ Now we will rewrite the bean configuration file by using **emp1** bean configuration to **emp2, emp3**.

applicationContext.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans
3.     xmlns="http://www.springframework.org/schema/beans"
4.     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5.     xsi:schemaLocation="http://www.springframework.org/schema/beans
6.     http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
7.
8.     <bean id="emp1" class="com.neo.spring.bean.Employee">
9.           <property name="eno" value="1001"/>
10.                <property name="ename" value="sekhar"/>
11.                <property name="sal" value="2500"/>
12.                <property name="desig" value="teammember"/>
13.                <property name="hno" value="1-52-A"/>
14.                <property name="city" value="Gunipalli"/>
15.          </bean>
16.
17.          <bean id="emp2" class="com.neo.spring.bean.Employee"
18.                                                          parent="emp1">
19.                <property name="eno" value="1002"/>
20.                <property name="ename" value="somu"/>
21.                <property name="hno" value="1-52-B"/>
22.          </bean>
23.
24.          <bean id="emp3" class="com.neo.spring.bean.Employee"
```

```
25.                                                          parent="emp1">
26.              <property name="eno" value="1003"/>
27.              <property name="ename" value="somasekhar"/>
28.              <property name="hno" value="1-52-B"/>
29.         </bean>
30.   </beans>
```

Now run the Client.java, we will get the same output.

Output

```
1001
sekhar
2500.0
teammember
1-52-A
Gunipalli

1002
somu
2500.0
teammember
1-52-B
Gunipalli

1003
somasekhar
2500.0
teammember
1-52-B
Gunipalli
```

⇨ This is the way of reusing bean configuration.

Q.) What happens if both parent, child configuration has same property?
Ans: child property overrides parent property.

'abstract' attribute

⇨ Generally in the bean configuration level inheritance, we will define one abstract bean configuration, which contains common properties, so that all the specific bean configurations can extend that abstract bean. But no one should access that abstract bean configuration. For that purpose we use **abstract** attribute of **<bean>** tag.

⇨ Now we rewrite the above spring configuration file by make use of **abstract** attribute.

applicationContext.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans
3.    xmlns="http://www.springframework.org/schema/beans"
4.    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5.    xsi:schemaLocation="http://www.springframework.org/schema/beans
6.    http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
7.
8.    <bean id="abstractemp" class="com.neo.spring.bean.Employee"
9.                                                    abstract="true" >
10.             <property name="sal" value="2500"></property>
11.             <property name="desig" value="teammember"></property>
12.             <property name="city" value="Gunipalli"></property>
13.   </bean>
```

```
14.
15.
16.         <bean id="emp1" class="com.neo.spring.bean.Employee"
17.                                                 parent="abstractemp">
18.               <property name="eno" value="1001"></property>
19.               <property name="ename" value="sekhar"></property>
20.               <property name="hno" value="1-52-A"></property>
21.         </bean>
22.
23.         <bean id="emp2" class="com.neo.spring.bean.Employee"
24.                                                 parent="abstractemp">
25.               <property name="eno" value="1002"></property>
26.               <property name="ename" value="somu"></property>
27.               <property name="hno" value="1-52-B"></property>
28.         </bean>
29.
30.         <bean id="emp3" class="com.neo.spring.bean.Employee"
31.                                                   parent="abstractemp">
32.               <property name="eno" value="1003"></property>
33.               <property name="ename" value="somasekhar"></property>
34.               <property name="hno" value="1-52-B"></property>
35.         </bean>
36.   </beans>
```

Client.java

```
1. package com.neo.spring.client;
2. import org.springframework.context.support.AbstractApplicationContext;
3. import org.springframework.context.support.ClassPathXmlApplicationContext;
4. import com.neo.spring.bean.Employee;
5. public class Client {
6.    private static AbstractApplicationContext context=new
7.                                       ClassPathXmlApplicationContext(
8.                            "com/neo/spring/config/applicationContext.xml");
9.    public static void main(String[] args) {
10.             Employee employee1=(Employee)context.getBean("emp1");
11.             employee1.displayEmpDetails();
12.             System.out.println();
13.
14.             Employee employee2=(Employee)context.getBean("emp2");
15.             employee2.displayEmpDetails();
16.             System.out.println();
17.
18.             Employee employee3=(Employee)context.getBean("emp3");
19.             employee3.displayEmpDetails();
20.             System.out.println();
21.
22.             // Employee
23.              employee=(Employee)context.getBean("abstractemp");
24.             // employee.displayEmpDetails();
25.       }
26.   }
```

Output

```
1001
sekhar
```

```
2500.0
teammember
1-52-A
Gunipalli

1002
somu
2500.0
teammember
1-52-B
Gunipalli

1003
somasekhar
2500.0
teammember
1-52-B
Gunipalli
```

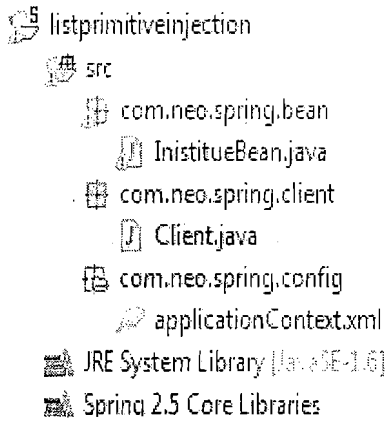
Exception in thread "main" org.springframework.beans.factory.BeanIsAbstractException: Error creating bean with name 'abstractemp': Bean definition is abstract

Note: If we will try to access abstract bean, we will get exception.

Injecting Collection objects

- If the dependencies are of type Arrays, List, Set, Map, properties...etc, The way we inject the values for these dependencies are different from normal injection
- In general for some of the frame work given beans we need to inject the Collection type objects.

Injecting List object (where list contains primitive types or String objects)



InistitueBean.java

```
1. package com.neo.spring.bean;
2. import java.util.Iterator;
3.  import java.util.List;
4.
5. public class InistitueBean {
6.    private String name;
7.    private List<String> facultyNames;
8.    private List<String> courseNames;
9.
10.         public void setName(String name) {
11.               this.name = name;
12.         }
13.
14.         public void setFacultyNames(List<String> facultyNames) {
15.               this.facultyNames = facultyNames;
16.         }
17.
18.         public void setCourseNames(List<String> courseNames) {
19.               this.courseNames = courseNames;
20.         }
21.
22.         public void displayDetails() {
23.               System.out.println("Institue Name : " + name);
24.               System.out.println("Faculty Names are ...");
25.               for (String facutlyName : facultyNames) {
26.                     System.out.println(facutlyName);
27.               }
28.               Double double1 = new Double(2500);
29.               System.out.println("Course names are ...");
30.               Iterator<String> iterator = courseNames.iterator();
31.               while (iterator.hasNext()) {
32.                     String courseName = iterator.next();
33.                     System.out.println(courseName);
34.               }
```

```
35.         }
36.   }
```

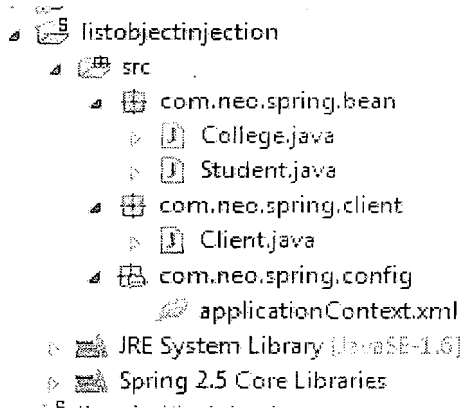
applicationContext.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans default-init-method="myInit" default-destroy-method="myDestroy"
3.     xmlns="http://www.springframework.org/schema/beans"
4.     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5.     xmlns:context="http://www.springframework.org/schema/context"
6.     xsi:schemaLocation="http://www.springframework.org/schema/beans
7.         http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
8.         http://www.springframework.org/schema/context
9.     http://www.springframework.org/schema/context/spring-context-2.5.xsd">
10.         <bean id="ibref" class="com.neo.spring.bean.InistitueBean">
11.               <property name="name">
12.                     <value>Naresh I Technologies</value>
13.               </property>
14.               <property name="facultyNames">
15.                     <list>
16.                           <value>sekhar</value>
17.                           <value>lnrao</value>
18.                           <value>narayana</value>
19.                     </list>
20.               </property>
21.
22.               <property name="courseNames">
23.                     <list>
24.                           <value>spring</value>
25.                           <value>struts</value>
26.                           <value>hibernate</value>
27.                     </list>
28.               </property>
29.
30.         </bean>
31.   </beans>
```

Client.java

```
1. package com.neo.spring.client;
2.
3. import org.springframework.context.support.AbstractApplicationContext;
4. import org.springframework.context.support.ClassPathXmlApplicationContext;
5. import com.neo.spring.bean.InistitueBean;
6.
7. public class Client {
8.
9.     private static AbstractApplicationContext context = new
10.                                     ClassPathXmlApplicationContext(
11.                     "com/neo/spring/config/applicationContext.xml");
12.
13.         public static void main(String[] args) {
14.               InistitueBean bean = (InistitueBean)
15.                                       context.getBean("ibref");
16.               bean.displayDetails();
17.         }
18.   }
```

Injecting List object (where list contains user defined object types)



Student.java

```
1. package com.neo.spring.bean;
2.
3. public class Student {
4.     private int sno;
5.     private String sname;
6.     private double fee;
7.
8.     public int getSno() {
9.           return sno;
10.          }
11.
12.          public void setSno(int sno) {
13.                this.sno = sno;
14.          }
15.
16.          public String getSname() {
17.                return sname;
18.          }
19.
20.          public void setSname(String sname) {
21.                this.sname = sname;
22.          }
23.
24.          public double getFee() {
25.                return fee;
26.          }
27.
28.          public void setFee(double fee) {
29.                this.fee = fee;
30.          }
31.
32.    }
33.
```

College.java

```
1. package com.neo.spring.bean;
2.
```

```
3. import java.util.Iterator;
4. import java.util.List;
5.
6. public class College {
7.    private String name;
8.    private List<Student> students;
9.
10.         public void setName(String name) {
11.               this.name = name;
12.         }
13.
14.         public void setStudents(List<Student> students) {
15.               this.students = students;
16.         }
17.
18.         public void displayDetails() {
19.               System.out.println("College Name is : " + name);
20.               System.out.println("Student details  are ....");
21.               Iterator<Student> iterator = students.iterator();
22.               while (iterator.hasNext()) {
23.                     Student student = iterator.next();
24.                     System.out.println(student.getSno());
25.                     System.out.println(student.getSname());
26.                     System.out.println(student.getFee());
27.                     System.out.println();
28.               }
29.
30.         }
31.
32.   }
```

applicationContext.java

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans default-init-method="myInit" default-destroy-method="myDestroy"
3.    xmlns="http://www.springframework.org/schema/beans"
4.    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5.    xmlns:context="http://www.springframework.org/schema/context"
6.    xsi:schemaLocation="http://www.springframework.org/schema/beans
7.        http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
8.        http://www.springframework.org/schema/context
9.     http://www.springframework.org/schema/context/spring-context-2.5.xsd">
10.       <bean id="s1" class="com.neo.spring.bean.Student" >
11.             <property name="sno" value="1001"></property>
12.             <property name="sname">
13.                   <value>sekhar</value>
14.             </property>
15.             <property name="fee" value="2500"></property>
16.       </bean>
17.
18.       <bean id="s2" class="com.neo.spring.bean.Student" >
19.             <property name="sno" value="1002"></property>
20.             <property name="sname" value="somu"></property>
21.             <property name="fee" value="5900"></property>
22.       </bean>
23.
```

```
24.         <bean id="s3" class="com.neo.spring.bean.Student">
25.               <property name="sno" value="1003"></property>
26.               <property name="sname" value="somasekhar"></property>
27.               <property name="fee" value="4200"></property>
28.         </bean>
29.
30.         <bean id="collegeNameRef" class="java.lang.String">
31.               <constructor-arg value="Sri Indhu College" />
32.         </bean>
33.
34.         <bean id="college" class="com.neo.spring.bean.College">
35.               <property name="name" ref="collegeNameRef"></property>
36.               <property name="students">
37.                     <list>
38.                           <ref bean="s1" />
39.                           <ref bean="s2" />
40.                           <ref bean="s3" />
41.                     </list>
42.               </property>
43.         </bean>
44.
45.   </beans>
```

Client.java

```
1. package com.neo.spring.client;
2. import org.springframework.context.support.AbstractApplicationContext;
3. import org.springframework.context.support.ClassPathXmlApplicationContext;
4. import com.neo.spring.bean.College;
5. public class Client {
6.
7.    private static AbstractApplicationContext context = new
8.                                 ClassPathXmlApplicationContext(
9.               "com/neo/spring/config/applicationContext.xml");
10.
11.         public static void main(String[] args) {
12.               College college = (College) context.getBean("college");
13.               college.displayDetails();
14.         }
15.   }
```

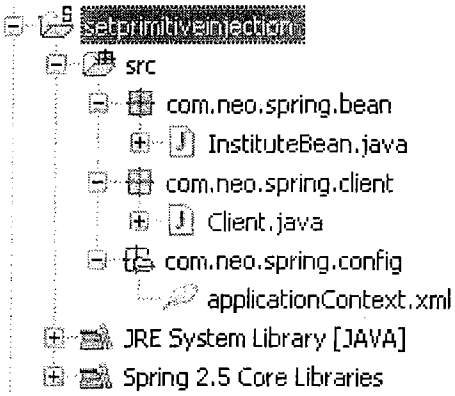
Output

```
College Name is : Sri Indhu college
Student details  are ....
1001
sekhar
2500.0

1002
somu
5900.0

1003
somasekhar
4200.0
```

Set Primitive Injection



InstituteBean.java

```
1. package com.neo.spring.bean;
2. import java.util.Iterator;
3. import java.util.Set;
4. public class InstituteBean {
5.    private String name;
6.    private Set<String> facultyNames;
7.    private Set<String> courseNames;
8.    public void setName(String name) {
9.          this.name = name;
10.         }
11.         public void setFacultyNames(Set<String> facultyNames) {
12.               this.facultyNames = facultyNames;
13.         }
14.         public void setCourseNames(Set<String> courseNames) {
15.               this.courseNames = courseNames;
16.         }
17.         public void dispalyDetails(){
18.               System.out.println("Institute name:"+name);
19.               System.out.println("faculty names....");
20.               for (String facultyName : facultyNames) {
21.                     System.out.println(facultyName);
22.               }
23.               System.out.println("course names....");
24.               Iterator<String> iterator=courseNames.iterator();
25.               while(iterator.hasNext()){
26.                     String courseName=iterator.next();
27.                     System.out.println(courseName);
28.               }
29.         }
30.   }
```

applicationContext.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans
3.    xmlns="http://www.springframework.org/schema/beans"
4.    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5.    xsi:schemaLocation="http://www.springframework.org/schema/beans
6.    http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
7.
```

```
8.    <bean id="ibref" class="com.neo.spring.bean.InstituteBean">
9.          <property name="name" value="Naresh I Technologies"></property>
10.               <property name="facultyNames">
11.                     <set>
12.                           <value>sekhar</value>
13.                           <value>lnrao</value>
14.                           <value>narayana</value>
15.                     </set>
16.               </property>
17.               <property name="courseNames">
18.                     <set>
19.                           <value>spring</value>
20.                           <value>struts</value>
21.                           <value>hibernate</value>
22.                     </set>
23.               </property>
24.         </bean>
25.   </beans>
```

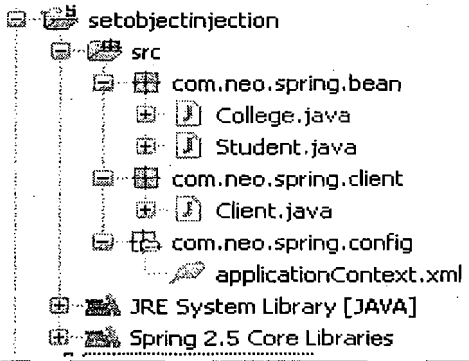
Client.java

```
1. package com.neo.spring.client;
2. import org.springframework.context.support.AbstractApplicationContext;
3. import org.springframework.context.support.ClassPathXmlApplicationContext;
4. import com.neo.spring.bean.InstituteBean;
5. public class Client {
6.    private static AbstractApplicationContext context=new
7.                                    ClassPathXmlApplicationContext(
8.                  "com/neo/spring/config/applicationContext.xml");
9.    public static void main(String[] args) {
10.               InstituteBean bean=(InstituteBean)context.getBean("ibref");
11.               bean.dispalyDetails();
12.         }
13.   }
```

output

```
Institute name:Naresh I Technologies
faculty names....
sekhar
lnrao
narayana
course names....
spring
struts
hibernate
```

Set Object Injection



Student.java

```
1. package com.neo.spring.bean;
2. public class Student {
3.     private int sno;
4.     private String sname;
5.     private String fee;
6.     public int getSno() {
7.           return sno;
8.     }
9.     public void setSno(int sno) {
10.          this.sno = sno;
11.    }
12.    public String getSname() {
13.          return sname;
14.    }
15.    public void setSname(String sname) {
16.          this.sname = sname;
17.    }
18.    public String getFee() {
19.          return fee;
20.    }
21.    public void setFee(String fee) {
22.          this.fee = fee;
23.    }
24. }
```

College.java

```
1. package com.neo.spring.bean;
2. import java.util.Iterator;
3. import java.util.Set;
4. public class College {
5.     private String name;
6.     private Set<Student> students;
7.     public void setName(String name) {
8.           this.name = name;
9.     }
10.          public void setStudents(Set<Student> students) {
11.                this.students = students;
12.          }
13.          public void displayDetails(){
14.                System.out.println("college name:"+name);
15.                System.out.println("Student details....");
16.                Iterator<Student> iterator=students.iterator();
17.                while(iterator.hasNext()){
18.                      Student student=iterator.next();
19.                      System.out.println(student.getSno());
20.                      System.out.println(student.getSname());
21.                      System.out.println(student.getFee());
22.                      System.out.println();
23.                }
24.          }
25.    }
```

applicationContext.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans
3.     xmlns="http://www.springframework.org/schema/beans"
4.     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5.     xsi:schemaLocation="http://www.springframework.org/schema/beans
6.     http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
7.
8.     <bean id="s1" class="com.neo.spring.bean.Student">
9.           <property name="sno" value="1001"></property>
10.                <property name="sname" value="sekhar"></property>
11.                <property name="fee" value="2500"></property>
12.          </bean>
13.
14.          <bean id="s2" class="com.neo.spring.bean.Student">
15.                <property name="sno" value="1002"></property>
16.                <property name="sname" value="somu"></property>
17.                <property name="fee" value="2500"></property>
18.          </bean>
19.
20.          <bean id="s3" class="com.neo.spring.bean.Student">
21.                <property name="sno" value="1003"></property>
22.                <property name="sname" value="somasekhar"></property>
23.                <property name="fee" value="2500"></property>
24.          </bean>
25.
26.          <bean id="collegeref" class="com.neo.spring.bean.College">
27.                <property name="name" value="Sri Indhu College"></property>
28.                <property name="students">
29.                      <set>
30.                            <ref bean="s1"/>
31.                            <ref bean="s2"/>
32.                            <ref bean="s3"/>
33.                      </set>
34.                </property>
35.          </bean>
36.    </beans>
```

Client.java

```
1. package com.neo.spring.client;
2. import org.springframework.context.support.AbstractApplicationContext;
3. import org.springframework.context.support.ClassPathXmlApplicationContext;
4. import com.neo.spring.bean.College;
5. public class Client {
6.     private static AbstractApplicationContext context=new
7.                                 ClassPathXmlApplicationContext("
8.                 com/neo/spring/config/applicationContext.xml");
9.     public static void main(String[] args) {
10.                College college=(College)context.getBean("collegeref");
11.                college.displayDetails();
12.          }
13.    }
```

Output

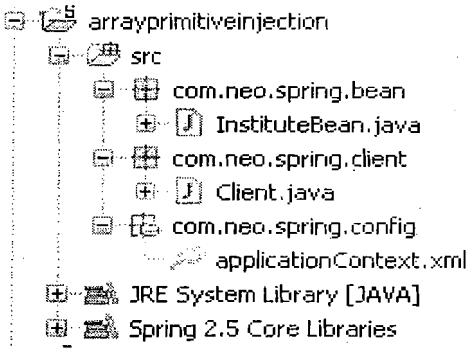
```
college name:Sri Indhu College
Student details....
```

```
1001
sekhar
2500

1002
somu
2500

1003
somasekhar
2500
```

Array Primitive Injection



InstituteBean.java

```
1. package com.neo.spring.bean;
2. public class InstituteBean {
3.     private String name;
4.     private String[] facultyNames;
5.     private String[] courseNames;
6.     public void setName(String name) {
7.           this.name = name;
8.     }
9.     public void setFacultyNames(String[] facultyNames) {
10.          this.facultyNames = facultyNames;
11.    }
12.    public void setCourseNames(String[] courseNames) {
13.          this.courseNames = courseNames;
14.    }
15.    public void dispalyDetails(){
16.          System.out.println("Institute name:"+name);
17.          System.out.println("faculty names....");
18.          for (String facultyName : facultyNames) {
19.                System.out.println(facultyName);
20.          }
21.          System.out.println("course names....");
22.          for (String courseName : courseNames) {
23.                System.out.println(courseName);
24.          }
25.    }
26. }
```

applicationContext.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans
```

```
3.    xmlns="http://www.springframework.org/schema/beans"
4.    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5.    xsi:schemaLocation="http://www.springframework.org/schema/beans
6.    http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
7.
8.    <bean id="ibref" class="com.neo.spring.bean.InstituteBean">
9.          <property name="name" value="Naresh I Technologies"></property>
10.               <!-- <property name="facultyNames">
11.                     <set>
12.                           <value>sekhar</value>
13.                           <value>lnrao</value>
14.                           <value>narayana</value>
15.                     </set>
16.               </property>
17.               <property name="courseNames">
18.                     <set>
19.                           <value>spring</value>
20.                           <value>struts</value>
21.                           <value>hibernate</value>
22.                     </set>
23.               </property> -->
24.               <property name="facultyNames">
25.                     <list>
26.                           <value>sekhar</value>
27.                           <value>lnrao</value>
28.                           <value>narayana</value>
29.                     </list>
30.               </property>
31.               <property name="courseNames">
32.                     <list>
33.                           <value>spring</value>
34.                           <value>struts</value>
35.                           <value>hibernate</value>
36.                     </list>
37.               </property>
38.         </bean>
39.   </beans>
```

Client.java

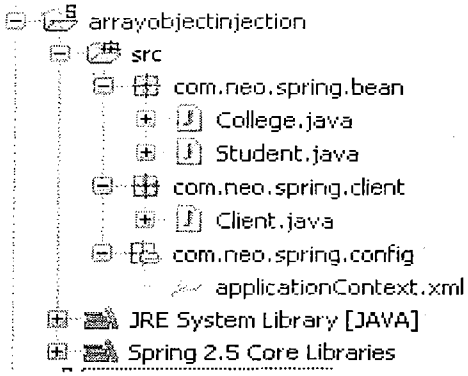
```
1. package com.neo.spring.client;
2. import org.springframework.context.support.AbstractApplicationContext;
3. import org.springframework.context.support.ClassPathXmlApplicationContext;
4. import com.neo.spring.bean.InstituteBean;
5. public class Client {
6.    private static AbstractApplicationContext context=new
7.                                    ClassPathXmlApplicationContext(
8.                              "com/neo/spring/config/applicationContext.xml");
9.    public static void main(String[] args) {
10.               InstituteBean bean=(InstituteBean)context.getBean("ibref");
11.               bean.dispalyDetails();
12.         }
13.   }
```

Output

```
Institute name:Naresh I Technologies
```

```
faculty names....
sekhar
lnrao
narayana
course names....
spring
struts
hibernate
```

Array Object Injection



Student.java

```
1. package com.neo.spring.bean;
2. public class Student {
3.     private int sno;
4.     private String sname;
5.     private String fee;
6.     public int getSno() {
7.           return sno;
8.     }
9.     public void setSno(int sno) {
10.          this.sno = sno;
11.    }
12.    public String getSname() {
13.          return sname;
14.    }
15.    public void setSname(String sname) {
16.          this.sname = sname;
17.    }
18.    public String getFee() {
19.          return fee;
20.    }
21.    public void setFee(String fee) {
22.          this.fee = fee;
23.    }
24. }
```

College.java

```
1. package com.neo.spring.bean;
2. public class College {
3.     private String name;
4.     private Student[] students;
5.     public void setName(String name) {
```

```
6.          this.name = name;
7.    }
8.    public void setStudents(Student[] students) {
9.          this.students = students;
10.         }
11.         public void displayDetails(){
12.               System.out.println("college name:"+name);
13.               System.out.println("Student details....");
14.               for (Student student : students) {
15.                     System.out.println(student.getSno());
16.                     System.out.println(student.getSname());
17.                     System.out.println(student.getFee());
18.                     System.out.println();
19.               }
20.         }
21.   }
```

applicationContext.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans
3.    xmlns="http://www.springframework.org/schema/beans"
4.    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5.    xsi:schemaLocation="http://www.springframework.org/schema/beans
6.    http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
7.
8.    <bean id="s1" class="com.neo.spring.bean.Student">
9.          <property name="sno" value="1001"></property>
10.               <property name="sname" value="sekhar"></property>
11.               <property name="fee" value="2500"></property>
12.         </bean>
13.
14.         <bean id="s2" class="com.neo.spring.bean.Student">
15.               <property name="sno" value="1002"></property>
16.               <property name="sname" value="somu"></property>
17.               <property name="fee" value="2500"></property>
18.         </bean>
19.
20.         <bean id="s3" class="com.neo.spring.bean.Student">
21.               <property name="sno" value="1003"></property>
22.               <property name="sname" value="somasekhar"></property>
23.               <property name="fee" value="2500"></property>
24.         </bean>
25.
26.         <bean id="collegeref" class="com.neo.spring.bean.College">
27.               <property name="name" value="Sri Indhu College"></property>
28.               <property name="students">
29.                     <!-- <set>
30.                           <ref bean="s1"/>
31.                           <ref bean="s2"/>
32.                           <ref bean="s3"/>
33.                     </set> -->
34.                     <list>
35.                           <ref bean="s1"/>
36.                           <ref bean="s2"/>
37.                           <ref bean="s3"/>
```

```
38.                     </list>
39.                 </property>
40.         </bean>
41.   </beans>
```

Client.java

```
1. package com.neo.spring.client;
2. import org.springframework.context.support.AbstractApplicationContext;
3. import org.springframework.context.support.ClassPathXmlApplicationContext;
4. import com.neo.spring.bean.College;
5. public class Client {
6.     private static AbstractApplicationContext context=new
7.                                     ClassPathXmlApplicationContext(
8.                   "com/neo/spring/config/applicationContext.xml");
9.     public static void main(String[] args) {
10.                College college=(College)context.getBean("collegeref");
11.                college.displayDetails();
12.         }
13.   }
```

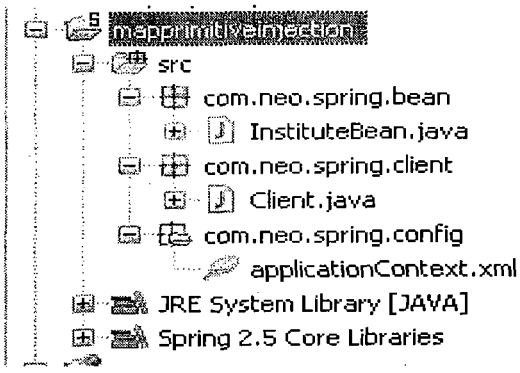
Output

```
college name:Sri Indhu College
Student details....
1001
sekhar
2500

1002
somu
2500

1003
somasekhar
2500
```

Map Primitive Injection



InstituteBean.java

```
1. package com.neo.spring.bean;
2. import java.util.Iterator;
3. import java.util.Map;
4. import java.util.Set;
5. public class InstituteBean {
6.     private String name;
```

```
7.    private Map<Integer, String> facultyMap;
8.    private Map<Integer, String> courseMap;
9.    public void setName(String name) {
10.             this.name = name;
11.       }
12.       public void setFacultyMap(Map<Integer, String> facultyMap) {
13.             this.facultyMap = facultyMap;
14.       }
15.       public void setCourseMap(Map<Integer, String> courseMap) {
16.             this.courseMap = courseMap;
17.       }
18.       public void dispalyDetails(){
19.             System.out.println("Institute name:"+name);
20.             System.out.println("faculty names.....");
21.             Set<Integer> facultyKey=facultyMap.keySet();
22.             Iterator<Integer> iterator1=facultyKey.iterator();
23.             while (iterator1.hasNext()) {
24.                   Integer facultyId=iterator1.next();
25.                   String facultyName=facultyMap.get(facultyId);
26.                   System.out.println(facultyId+"-->"+facultyName);
27.             }
28.             System.out.println("course details .....");
29.             Set<Integer> courseKey=courseMap.keySet();
30.             Iterator<Integer> iterator2=courseKey.iterator();
31.             while (iterator2.hasNext()) {
32.                   Integer courseId =  iterator2.next();
33.                   String courseName=courseMap.get(courseId);
34.                   System.out.println(courseId+"-->"+courseName);
35.             }
36.       }
37.   }
```

applicationContext.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans
3.    xmlns="http://www.springframework.org/schema/beans"
4.    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5.    xsi:schemaLocation="http://www.springframework.org/schema/beans
6.    http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
7.
8.    <bean id="ibref" class="com.neo.spring.bean.InstituteBean">
9.          <property name="name" value="Naresh I Technologies"></property>
10.             <property name="facultyMap">
11.                   <map>
12.                         <entry key="1001" value="sekhar"></entry>
13.                         <entry key="1002" value="1nrao"></entry>
14.                         <entry key="1003" value="narayana"></entry>
15.                   </map>
16.             </property>
17.             <property name="courseMap">
18.                   <map>
19.                         <entry key="1111">
20.                               <value>spring</value>
21.                         </entry>
22.                         <entry key="1112">
```

```
23.                                 <value>struts</value>
24.                             </entry>
25.                             <entry key="1113">
26.                                 <value>hibernate</value>
27.                             </entry>
28.                         </map>
29.                 </property>
30.         </bean>
31.     </beans>
```

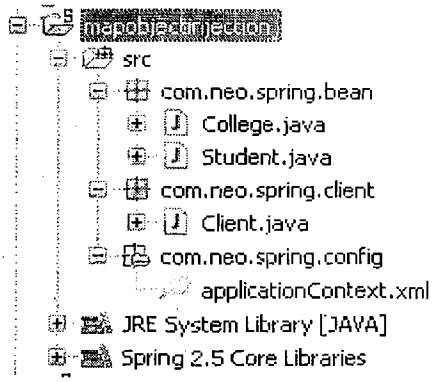
Client.java

```
1. package com.neo.spring.client;
2. import org.springframework.context.support.AbstractApplicationContext;
3. import org.springframework.context.support.ClassPathXmlApplicationContext;
4. import com.neo.spring.bean.InstituteBean;
5. public class Client {
6.     private static AbstractApplicationContext context=new
7.                                         ClassPathXmlApplicationContext(
8.                                 "com/neo/spring/config/applicationContext.xml");
9.     public static void main(String[] args) {
10.                 InstituteBean bean=(InstituteBean)context.getBean("ibref");
11.                 bean.dispalyDetails();
12.         }
13.     }
```

Output

```
Institute name:Naresh I Technologies
faculty names.....
1001-->sekhar
1002-->lnrao
1003-->narayana
course details .....
1111-->spring
1112-->struts
1113-->hibernate
```

Map Object Injection



Student.java

```
1. package com.neo.spring.bean;
2. public class Student {
3.     private int sno;
4.     private String sname;
5.     private String fee;
```

```
6.    public int getSno() {
7.          return sno;
8.    }
9.    public void setSno(int sno) {
10.               this.sno = sno;
11.         }
12.         public String getSname() {
13.               return sname;
14.         }
15.         public void setSname(String sname) {
16.               this.sname = sname;
17.         }
18.         public String getFee() {
19.               return fee;
20.         }
21.         public void setFee(String fee) {
22.               this.fee = fee;
23.         }
24.   }
```

College.java

```
1. package com.neo.spring.bean;
2. import java.util.Iterator;
3. import java.util.Map;
4. import java.util.Set;
5. public class College {
6.    private String name;
7.    private Map<Integer, Student> studentMap;
8.    public void setName(String name) {
9.          this.name = name;
10.         }
11.
12.         public void setStudentMap(Map<Integer, Student> studentMap) {
13.               this.studentMap = studentMap;
14.         }
15.
16.         public void displayDetails(){
17.               System.out.println("college name:"+name);
18.               System.out.println("Student details....");
19.               Set<Integer> studentKey=studentMap.keySet();
20.               Iterator<Integer> iterator=studentKey.iterator();
21.               while(iterator.hasNext()){
22.                     Integer studentId=iterator.next();
23.                     System.out.println(studentId);
24.                     Student student=studentMap.get(studentId);
25.                     System.out.println(student.getSno());
26.                     System.out.println(student.getSname());
27.                     System.out.println(student.getFee());
28.                     System.out.println();
29.               }
30.         }
31.   }
```

applicationContext.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
```

```
2. <beans
3.     xmlns="http://www.springframework.org/schema/beans"
4.     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5.     xsi:schemaLocation="http://www.springframework.org/schema/beans
6.     http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
7.
8.     <bean id="s1" class="com.neo.spring.bean.Student">
9.           <property name="sno" value="1001"></property>
10.                <property name="sname" value="sekhar"></property>
11.                <property name="fee" value="2500"></property>
12.          </bean>
13.
14.          <bean id="s2" class="com.neo.spring.bean.Student">
15.                <property name="sno" value="1002"></property>
16.                <property name="sname" value="somu"></property>
17.                <property name="fee" value="2500"></property>
18.          </bean>
19.
20.          <bean id="s3" class="com.neo.spring.bean.Student">
21.                <property name="sno" value="1003"></property>
22.                <property name="sname" value="somasekhar"></property>
23.                <property name="fee" value="2500"></property>
24.          </bean>
25.
26.          <bean id="collegeref" class="com.neo.spring.bean.College">
27.                <property name="name" value="Sri Indhu College"></property>
28.                <property name="studentMap">
29.                     <map>
30.                          <entry key="1111" value-ref="s1">
31.                          </entry>
32.                          <entry key="1112" >
33.                               <ref bean="s2"/>
34.                          </entry>
35.                               <entry key="1113">
36.                               <ref bean="s3"/>
37.                          </entry>
38.                     </map>
39.                </property>
40.          </bean>
41.    </beans>
```

Client.java

```
1. package com.neo.spring.client;
2. import org.springframework.context.support.AbstractApplicationContext;
3. import org.springframework.context.support.ClassPathXmlApplicationContext;
4. import com.neo.spring.bean.College;
5. public class Client {
6.     private static AbstractApplicationContext context=new
7.                     ClassPathXmlApplicationContext(
8.                     "com/neo/spring/config/applicationContext.xml");
9.     public static void main(String[] args) {
10.                College college=(College)context.getBean("collegeref");
11.                college.displayDetails();
12.          }
13.    }
```

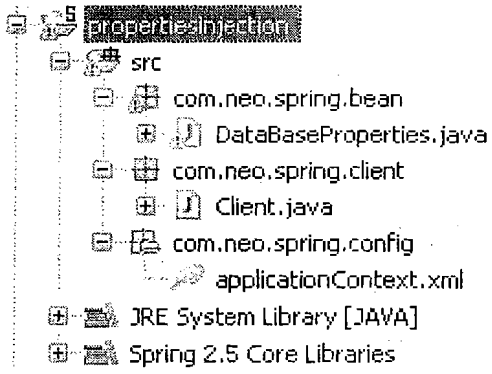
Output

```
college name:Sri Indhu College
Student details....
1111
1001
sekhar
2500

1112
1002
somu
2500

1113
1003
somasekhar
2500
```

Properties Primitive Injection



DataBaseProperties.java

```
1. package com.neo.spring.bean;
2. import java.util.Enumeration;
3. import java.util.Properties;
4. public class DataBaseProperties {
5.     private Properties dbProperties;
6.     public void setDbProperties(Properties dbProperties) {
7.           this.dbProperties = dbProperties;
8.     }
9.     public void dispalyDetails(){
10.                Enumeration enumeration=dbProperties.keys();
11.                while (enumeration.hasMoreElements()) {
12.                      String id=(String)enumeration.nextElement();
13.                      String name=dbProperties.getProperty(id);
14.                      System.out.println(id+"-->"+name);
15.                }
16.          }
17.    }
```

applicationContext.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans
3.     xmlns="http://www.springframework.org/schema/beans"
4.     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
5.    xsi:schemaLocation="http://www.springframework.org/schema/beans
6.    http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
7.
8.    <bean id="ibref" class="com.neo.spring.bean.DataBaseProperties">
9.    <property name="dbProperties">
10.   <props>
11.    <prop key="driverClass">oracle.jdbc.driver.OracleDriver</prop>
12.    <prop key="username">scott</prop>
13.    <prop key="password">tiger</prop>
14.    <prop key="url">jdbc:oracle:thin:@localhost:1521:server</prop>
15.   </props>
16.   </property>
17.
18.   </bean>
19.   </beans>
```

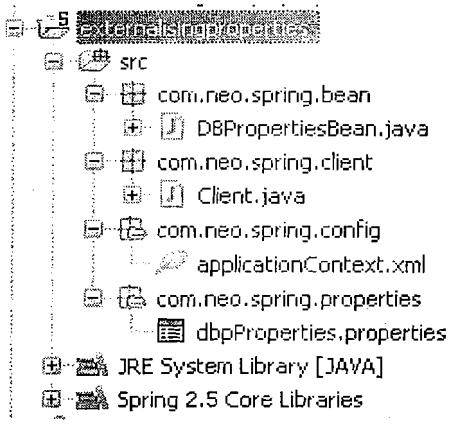
Client.java

```
1. package com.neo.spring.client;
2. import org.springframework.context.support.AbstractApplicationContext;
3. import org.springframework.context.support.ClassPathXmlApplicationContext;
4. import com.neo.spring.bean.DataBaseProperties;
5. public class Client {
6.    private static AbstractApplicationContext context=new
7.                            ClassPathXmlApplicationContext(
8.                      "com/neo/spring/config/applicationContext.xml");
9.    public static void main(String[] args) {
10.             DataBaseProperties bean=(DataBaseProperties)
11.                                                  context.getBean("ibref");
12.             bean.dispalyDetails();
13.       }
14.   }
```

Output

```
driverClass-->oracle.jdbc.driver.OracleDriver
url-->jdbc:oracle:thin:@localhost:1521:server
password-->tiger
username-->scott
```

Externilize properties from spring configuration file



DBPropertiesBean.java

```
1. package com.neo.spring.bean;
2. public class DBPropertiesBean {
3.     private String username;
4.     private String password;
5.     private String url;
6.     private String driverClass;
7.         public void setUsername(String username) {
8.                 this.username = username;
9.         }
10.        public void setPassword(String password) {
11.                this.password = password;
12.        }
13.        public void setUrl(String url) {
14.                this.url = url;
15.        }
16.        public void setDriverClass(String driverClass) {
17.                this.driverClass = driverClass;
18.        }
19.        public void displayDetails(){
20.                System.out.println(username);
21.                System.out.println(password);
22.                System.out.println(url);
23.                System.out.println(driverClass);
24.        }
25.    }
```

dbpProperties.properties

```
1. un=scott
2. pwd=tiger
3. url=jdbc:oracle:thin:@localhost:1521:server
4. dc=oracle.jdbc.driver.OracleDriver
```

applicationContext.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans
3.     xmlns="http://www.springframework.org/schema/beans"
4.     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5.     xsi:schemaLocation="http://www.springframework.org/schema/beans
```

```
6.    http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
7.
8.    <bean class="org.springframework.beans.factory.config.Property
9.                                                    PlaceholderConfigurer">
10.             <property name="location"
11.            value="com\neo\spring\properties\dbpProperties.properties"/>
12.       </bean>
13.
14.   <bean id="dbpropsref" class="com.neo.spring.bean.DBPropertiesBean">
15.             <property name="username" value="${un}"/>
16.             <property name="password" value="${pwd}"/>
17.             <property name="url" value="${url}"/>
18.             <property name="driverClass" value="${dc}"/>
19.       </bean>
20.   </beans>
```

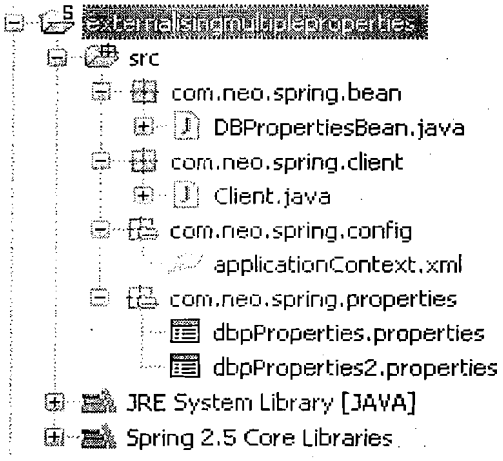
Client.java

```
1. package com.neo.spring.client;
2. import org.springframework.context.ApplicationContext;
3. import org.springframework.context.support.ClassPathXmlApplicationContext;
4. import com.neo.spring.bean.DBPropertiesBean;
5. public class Client {
6.    private static ApplicationContext context=new
7.                          ClassPathXmlApplicationContext(
8.                         "com/neo/spring/config/applicationContext.xml");
9.    public static void main(String[] args) {
10.             DBPropertiesBean bean=(DBPropertiesBean)
11.                                   context.getBean("dbpropsref");
12.             bean.displayDetails();
13.       }
14.   }
```

output

```
scott
tiger
jdbc:oracle:thin:@localhost:1521:server
oracle.jdbc.driver.OracleDriver
```

In above application we have only one properties file. So we have to take <property name="location" ---/>.
If we have two properties files then we have to take <property name="locations" ---/>. Observe the below application.



DBPropertiesBean.java

```
1. package com.neo.spring.bean;
2. public class DBPropertiesBean {
3.     private String username;
4.     private String password;
5.     private String url;
6.     private String driverClass;
7.     private String vendor;
8.     public void setVendor(String vendor) {
9.           this.vendor = vendor;
10.          }
11.          public void setUsername(String username) {
12.                this.username = username;
13.          }
14.          public void setPassword(String password) {
15.                this.password = password;
16.          }
17.          public void setUrl(String url) {
18.                this.url = url;
19.          }
20.          public void setDriverClass(String driverClass) {
21.                this.driverClass = driverClass;
22.          }
23.          public void displayDetails(){
24.                System.out.println(username);
25.                System.out.println(password);
26.                System.out.println(url);
27.                System.out.println(driverClass);
28.                System.out.println(vendor);
29.          }
30.    }
```

dbpProperties.properties

```
1. un=scott
2. pwd=tiger
3. url=jdbc:oracle:thin:@localhost:1521:server
4. dc=oracle.jdbc.driver.OracleDriver
```

dbpProperties2.properties

```
1. vendor=oracle
```

applicationContext.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans
3.     xmlns="http://www.springframework.org/schema/beans"
4.     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5.     xsi:schemaLocation="http://www.springframework.org/schema/beans
6.     http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
7.
8.     <bean class="org.springframework.beans.factory.config.Property
9.                                                 PlaceholderConfigurer">
10.              <property name="locations">
11.              <list>
12.         <value>com\neo\spring\properties\dbpProperties.properties</value>
13.         <value>com\neo\spring\properties\dbpProperties2.properties</value>
14.              </list>
15.              </property>
16.          </bean>
17.
18.    <bean id="dbpropsref" class="com.neo.spring.bean.DBPropertiesBean">
19.              <property name="username" value="${un}"/>
20.              <property name="password" value="${pwd}"/>
21.              <property name="url" value="${url}"/>
22.              <property name="driverClass" value="${dc}"/>
23.              <property name="vendor" value="${vendor}"/>
24.         </bean>
25.    </beans>
```

Client.java

```
1. package com.neo.spring.client;
2. import org.springframework.context.ApplicationContext;
3. import org.springframework.context.support.ClassPathXmlApplicationContext;
4. import com.neo.spring.bean.DBPropertiesBean;
5. public class Client {
6.     private static ApplicationContext context=new
7.                              ClassPathXmlApplicationContext(
8.                             "com/neo/spring/config/applicationContext.xml");
9.     public static void main(String[] args) {
10.              DBPropertiesBean bean=(DBPropertiesBean)
11.                                       context.getBean("dbpropsref");
12.              bean.displayDetails();
13.         }
14.    }
```

output

```
scott
tiger
jdbc:oracle:thin:@localhost:1521:server
oracle.jdbc.driver.OracleDriver
Oracle
```

dependency-check

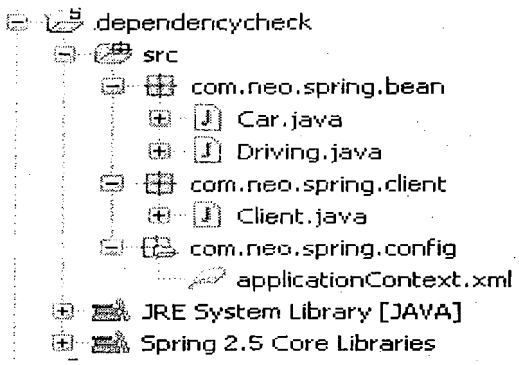
Q.) It is used to verify all dependencies of been that are configured via injection are injected or not?.

To implement dependency checking to a spring bean we make use of 'dependency-check' attribute of <bean> tag.

This attribute takes any one of the four following values.

1) **none**(it won't check whether dependencies injected or not)
2) **simple**(it checks all primitive dependencies injected or not)
3) **objects**(it checks all the object type dependencies injected or not)
4) **all**(it checks both primitives and object type dependencies injected or not)

If incase if we forgot to set any value either primitive or objective type it raise an exception called '**UnsatisfiedDependencyException**'.



Driving.java

```
1. package com.neo.spring.bean;
2. public class Driving {
3.     private String driver;
4.     public void setDriver(String driver) {
5.           this.driver = driver;
6.     }
7.     public String getDriver() {
8.           return driver;
9.     }
10.    }
```

Car.java

```
1. package com.neo.spring.bean;
2. public class Car {
3.     private String breaks;
4.     private Driving driving;
5.     public void setDriving(Driving driving) {
6.           this.driving = driving;
7.     }
8.     public void setBreaks(String breaks) {
9.           this.breaks = breaks;
10.          }
11.          public void driveCar(){
12.                System.out.println("old man come across the road");
13.                System.out.println("apply breaks");
14.                if (breaks!=null) {
15.                      System.out.println("old man saved");
16.                }else {
```

```
17.                     System.out.println("old man kicked the bucket");
18.               }
19.               if (driving.getDriver()!=null) {
20.                     System.out.println("driver is driving");
21.               }else {
22.                     System.out.println("owner is driving");
23.               }
24.         }
25.   }
```

applicationContext.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans
3.     xmlns="http://www.springframework.org/schema/beans"
4.     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5.     xsi:schemaLocation="http://www.springframework.org/schema/beans
6.     http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
7.
8.     <bean id="driveref" class="com.neo.spring.bean.Driving">
9.           <property name="driver" value="driver"></property>
10.          </bean>
11.          <bean id="carref" class="com.neo.spring.bean.Car"
12.                                                dependency-check="all">
13.               <property name="breaks" value="nice breaks"/>
14.               <property name="driving" ref="driveref"/>
15.          </bean>
16.   </beans>
```

Client.java

```
1. package com.neo.spring.client;
2. import org.springframework.context.ApplicationContext;
3. import org.springframework.context.support.ClassPathXmlApplicationContext;
4. import com.neo.spring.bean.Car;
5. public class Client {
6.     private static ApplicationContext context=new
7.                          ClassPathXmlApplicationContext(
8.                         "com/neo/spring/config/applicationContext.xml");
9.     public static void main(String[] args) {
10.               Car bean=(Car)context.getBean("carref");
11.               bean.driveCar();
12.          }
13.   }
```

output

```
old man come across the road
apply breaks
old man saved
driver is driving
```

Now observe below applicationContext.xml, here we are not configuring Car class property and giving **dependency-check="simple".**

applicationContext.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans
```

```
3.    xmlns="http://www.springframework.org/schema/beans"
4.    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5.    xsi:schemaLocation="http://www.springframework.org/schema/beans
6.    http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
7.
8.    <bean id="driveref" class="com.neo.spring.bean.Driving">
9.          <property name="driver" value="driver"></property>
10.         </bean>
11.         <bean id="carref" class="com.neo.spring.bean.Car"
12.                                                  dependency-check="simple">
13.               <property name="driving" ref="driveref"/>
14.         </bean>
15.   </beans>
```

Now run the client program and observe we'll get exception i.e., `UnsatisfiedDependencyException`.
If we are not configuring the Driving class property and giving **dependency-check="objects"**.

applicationContext.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans
3.    xmlns="http://www.springframework.org/schema/beans"
4.    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5.    xsi:schemaLocation="http://www.springframework.org/schema/beans
6.    http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
7.
8.    <bean id="driveref" class="com.neo.spring.bean.Driving">
9.          <property name="driver" value="driver"></property>
10.         </bean>
11.         <bean id="carref" class="com.neo.spring.bean.Car"
12.                                          dependency-check="objects">
13.               <property name="breaks" value="nice breaks"/>
14.         </bean>
15.   </beans>
```

Now run the client program and observe we'll get exception i.e., `UnsatisfiedDependencyException`.

Global default dependency checking

Explicitly define the dependency checking mode for every beans is tedious and error prone, you can set a default-dependency-check attribute in the <beans> root element to force the entire beans declared within <beans> root element to apply this rule. However, this root default mode will be overridden by a bean's own mode if specified.

applicationContext.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans
3.    xmlns="http://www.springframework.org/schema/beans"
4.    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5.    xsi:schemaLocation="http://www.springframework.org/schema/beans
6.    http://www.springframework.org/schema/beans/spring-beans-2.5.xsd"
7.    default-dependency-check="all">
8.
9.    <bean id="driveref" class="com.neo.spring.bean.Driving">
10.         <property name="driver" value="driver"></property>
```

```
11.         </bean>
12.         <bean id="carref" class="com.neo.spring.bean.Car" >
13.               <property name="breaks" value="nice breaks"/>
14.         </bean>
15.   </beans>
```

All beans declared in this configuration file are default to 'all' dependency checking mode.

@Required Annotation
In most scenarios, you just need to make sure a particular property has been set, but not all properties of a certain types (primitive, collection or object). The @Required Annotation can enforce this checking,

Spring dependency checking with @Required Annotation

Spring's dependency checking in bean configuration file is used to make sure all properties of a certain types (primitive, collection or object) have been set. In most scenarios, you just need to make sure a particular property has been set, but not all properties..

For this case, you need @Required annotation.

Simply apply the @Required annotation will not enforce the property checking, you also need to register an RequiredAnnotationBeanPostProcessor to aware of the @Required annotation in bean configuration file.

The RequiredAnnotationBeanPostProcessor can be enabled in two ways.

1. Include <context:annotation-config />

Add Spring context and <context:annotation-config /> in bean configuration file.

```
<beans
        ...
        xmlns:context="http://www.springframework.org/schema/context"
        ...
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-2.5.xsd">
        ...
        <context:annotation-config />
        ...
</beans>
```

2. Include RequiredAnnotationBeanPostProcessor

Include 'RequiredAnnotationBeanPostProcessor' directly in bean configuration file.

```
<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-2.5.xsd">
```

```
<bean
class="org.springframework.beans.factory.annotation.RequiredAnnotationBeanPostProcessor"/>

.....
.....
</beans>
```

@Required example

Driving.java

```
1. package com.neo.spring.bean;
2. public class Driving {
3.     private String driver;
4.     public void setDriver(String driver) {
5.           this.driver = driver;
6.     }
7.     public String getDriver() {
8.           return driver;
9.     }
10.    }
```

Car.java

```
1. package com.neo.spring.bean;
2. import org.springframework.beans.factory.annotation.Required;
3. public class Car {
4.     private String breaks;
5.     private Driving driving;
6.     @Required
7.     public void setDriving(Driving driving) {
8.           this.driving = driving;
9.     }
10.          public void setBreaks(String breaks) {
11.                this.breaks = breaks;
12.          }
13.          public void driveCar(){
14.                System.out.println("old man come across the road");
15.                System.out.println("apply breaks");
16.                if (breaks!=null) {
17.                      System.out.println("old man saved");
18.                }else {
19.                      System.out.println("old man kicked the bucket");
20.                }
21.                if (driving.getDriver()!=null) {
22.                      System.out.println("driver is driving");
23.                }else {
24.                      System.out.println("owner is driving");
25.                }
26.          }
27.    }
```

applicationContext.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans
3.     xmlns="http://www.springframework.org/schema/beans"
```

```
4.    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5.    xsi:schemaLocation="http://www.springframework.org/schema/beans
6.    http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
7.
8.    <bean id="driveref" class="com.neo.spring.bean.Driving">
9.          <property name="driver" value="driver"></property>
10.         </bean>
11.         <bean id="carref" class="com.neo.spring.bean.Car" >
12.               <property name="breaks" value="nice breaks"/>
13.         </bean>
14.   </beans>
```

Client.java

```
1. package com.neo.spring.client;
2. import org.springframework.context.ApplicationContext;
3. import org.springframework.context.support.ClassPathXmlApplicationContext;
4. import com.neo.spring.bean.Car;
5. public class Client {
6.    private static ApplicationContext context=new
7.                          ClassPathXmlApplicationContext(
8.                         "com/neo/spring/config/applicationContext.xml");
9.    public static void main(String[] args) {
10.             Car bean=(Car)context.getBean("carref");
11.             bean.driveCar();
12.         }
13.   }
```

Output:

```
org.springframework.beans.factory.BeanInitializationException:
```

NOTE: It is because driving property is unset.

AutoWiring

- **Auto wiring** means automatically injecting the dependencies.
- Instead of manually configuring the injection we done it automatically by using auto wiring
- To implement autowiring we use **autowire** attribute of <bean> tag.

There are five possible values to the **autowire** attribute of **<bean >**tag.

- no
- constructor
- byName
- byType
- autodetect

no: It won't allow autowiring.

byName:

Mainly it checks for 3 conditions if all these are valid then it injects the values by setter approach.

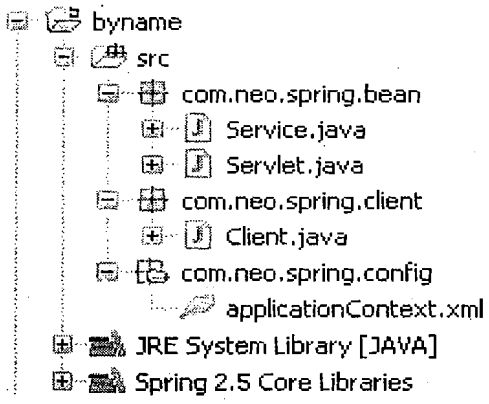
1. Dependency bean name
2. Configured bean name
3. Setter method name

- If dependency name is **"abc",** bean configuration should be **"abc"** and setter name method should be **"setAbc(Abc abc)"**

- When it finds **'autowire=byName'** for any bean configuration, then it first checks for dependency bean name in the dependent bean, then it will checks weather any bean is configured in the in the spring configuration file with the same name. If it finds then it will call corresponding setter method of the dependent bean.

Note: It doesn't bother about the argument types, it only bothers the names because it is autowiring **byName.**

Auto wiring byName



Service.java

```
1. package com.neo.spring.bean;
2. public class Service {
3.     public void serviceMethod(){
4.           System.out.println("Service.serviceMethod() called");
5.     }
6. }
```

Servlet.java

```
1. package com.neo.spring.bean;
2. public class Servlet {
3.    private Service service;
4.    public void setService(Service service) {
5.          this.service = service;
6.          System.out.println("injection happened");
7.    }
8.    public void servletMethod(){
9.          System.out.println("Servlet.servletMethod() called");
10.               service.serviceMethod();
11.         }
12.   }
```

applicationContext.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3.    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.    xsi:schemaLocation="http://www.springframework.org/schema/beans
5.    http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
6.
7.    <bean id="service" class="com.neo.spring.bean.Service"/>
8.    <bean id="servletref" class="com.neo.spring.bean.Servlet"
9.                                                    autowire="byName"/>
10.   </beans>
```

Client.java

```
1. package com.neo.spring.client;
2. import org.springframework.context.ApplicationContext;
3. import org.springframework.context.support.ClassPathXmlApplicationContext;
4.
5. import com.neo.spring.bean.Servlet;
6. public class Client {
7.    private static ApplicationContext context=new
8.                            ClassPathXmlApplicationContext(
9.                           "com/neo/spring/config/applicationContext.xml");
10.         public static void main(String[] args) {
11.               Servlet servlet=(Servlet)context.getBean("servletref");
12.               servlet.servletMethod();
13.         }
14.   }
```

output

```
injection happened
Servlet.servletMethod() called
Service.serviceMethod() called
```

Auto wiring (byType)

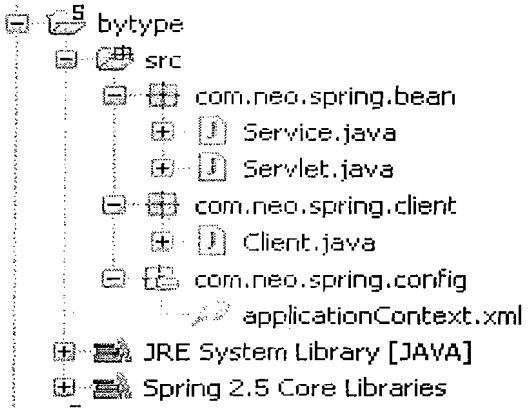
byType mainly it checks for 3 conditions. if all these are valid then it injects the values by setter approach.

1. **Dependency bean Type**
2. **Configured bean Type**
3. **Setter method Argument Type**

- When it finds **'autowire=byType'** for any bean configuration, then it first checks for dependency bean Type in the dependent bean, then it will checks the spring configuration file, weather any bean is configured with the dependency type, if it finds it will check for any setter method that takes bean dependency type as an argument. if it finds, then it will call that setter method.

Note: it doesn't bother about the setter method names, it only bothers the Types.

autowiring byType



Service.java

```
1. package com.neo.spring.bean;
2. public class Service {
3.     public void serviceMethod(){
4.           System.out.println("Service.serviceMethod() called");
5.     }
6. }
```

Servlet.java

```
1. package com.neo.spring.bean;
2. public class Servlet {
3.     private Service service1;
4.     public void setService3(Service service) {
5.           this.service = service;
6.           System.out.println("injection happened");
7.     }
8.     public void servletMethod(){
9.           System.out.println("Servlet.servletMethod() called");
10.                service.serviceMethod();
11.          }
12.    }
```

applicationContext.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans
3.     xmlns="http://www.springframework.org/schema/beans"
4.     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5.     xsi:schemaLocation="http://www.springframework.org/schema/beans
6.     http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
7.
```

```
8.    <bean id="service2" class="com.neo.spring.bean.Service"/>
9.    <bean id="servletref" class="com.neo.spring.bean.Servlet"
10.                                                    autowire="byType"/>
11.   </beans>
```

Client.java

```
1. package com.neo.spring.client;
2. import org.springframework.context.ApplicationContext;
3. import org.springframework.context.support.ClassPathXmlApplicationContext;
4. import com.neo.spring.bean.Servlet;
5. public class Client {
6.    private static ApplicationContext context=new
7.                  ClassPathXmlApplicationContext(
8.                 "com/neo/spring/config/applicationContext.xml");
9.    public static void main(String[] args) {
10.             Servlet servlet=(Servlet)context.getBean("servletref");
11.             servlet.servletMethod();
12.       }
13.   }
```

output

```
injection happened
Servlet.servletMethod() called
Service.serviceMethod() called
```

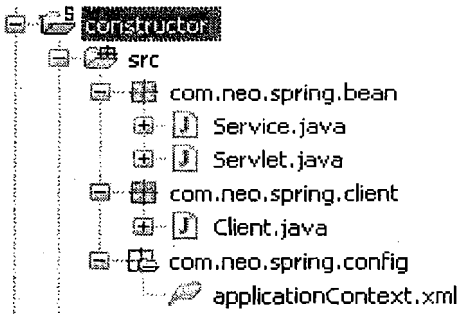
Auto wiring (constructor)

constructor mainly it checks for 3 conditions. if all these are valid then it injects the values by constructor approach.

- Dependency bean Type
- Configured bean Type
- Constructor Argument Type

➢ When it finds 'autowire= constructor' for any bean configuration, then it first checks for dependency bean Type in the dependent bean, then it will checks the spring configuration file, weather any bean is configured with the dependency type, if it finds it will check for constructor which will takes dependency type as an argument. If it then it will call the corresponding constructor.

Note: It won't bather about dependency name. It bathers about only dependency type, constructor argument type.



Service.java

```
1. package com.neo.spring.bean;
2. public class Service {
```

```
3.    public void serviceMethod(){
4.          System.out.println("Service.serviceMethod() called");
5.    }
6. }
```

Servlet.java

```
1. package com.neo.spring.bean;
2. public class Servlet {
3.    private Service service1;
4.    public Servlet(Service service) {
5.          this.service=service;
6.          System.out.println("Servlet constructor");
7.    }
8.    public void servletMethod(){
9.          System.out.println("Servlet.servletMethod() called");
10.               service.serviceMethod();
11.         }
12.   }
```

applicationContext.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans
3.    xmlns="http://www.springframework.org/schema/beans"
4.    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5.    xsi:schemaLocation="http://www.springframework.org/schema/beans
6.    http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
7.
8.    <bean id="service" class="com.neo.spring.bean.Service"/>
9.    <bean id="servletref" class="com.neo.spring.bean.Servlet"
10.                                          autowire="constructor"/>
11.   </beans>
```

Client.java

```
1. package com.neo.spring.client;
2. import org.springframework.context.ApplicationContext;
3. import org.springframework.context.support.ClassPathXmlApplicationContext;
4. import com.neo.spring.bean.Servlet;
5. public class Client {
6.    private static ApplicationContext context=new
7.                        ClassPathXmlApplicationContext(
8.                        "com/neo/spring/config/applicationContext.xml");
9.    public static void main(String[] args) {
10.             Servlet servlet=(Servlet)context.getBean("servletref");
11.             servlet.servletMethod();
12.         }
13.   }
```

output

```
Servlet constructor
Servlet.servletMethod() called
Service.serviceMethod() called
```

Auto wiring (autodetect)

autodetect Chooses "constructor" or "byType" through introspection of the bean class. If a default constructor is found, "byType" gets applied. If not found it will apply 'constructor'.

Service.java

```
1. package com.neo.spring.bean;
2. public class Service {
3.     public void serviceMethod(){
4.           System.out.println("Service.serviceMethod() called");
5.     }
6. }
```

Servlet.java

```
1. package com.neo.spring.bean;
2. public class Servlet {
3.     private Service service;
4.     public Servlet(Service service) {
5.           this.service=service;
6.           System.out.println("Servlet constructor");
7.     }
8.     public void setService(Service service) {
9.           this.service = service;
10.          System.out.println("setter method");
11.    }
12.    public void servletMethod(){
13.          System.out.println("Servlet.servletMethod() called");
14.          service.serviceMethod();
15.    }
16.  }
```

applicationContext.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans
3.     xmlns="http://www.springframework.org/schema/beans"
4.     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5.     xsi:schemaLocation="http://www.springframework.org/schema/beans
6.     http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
7.
8.     <bean id="service" class="com.neo.spring.bean.Service"></bean>
9.     <bean id="servletref" class="com.neo.spring.bean.Servlet"
10.                                          autowire="autodetect">
11.          </bean>
12.    </beans>
```

Client.java

```
1. package com.neo.spring.client;
2. import org.springframework.context.ApplicationContext;
3. import org.springframework.context.support.ClassPathXmlApplicationContext;
4. import com.neo.spring.bean.Servlet;
5. public class Client {
6.     private static ApplicationContext context=new
7.                          ClassPathXmlApplicationContext(
8.                         "com/neo/spring/config/applicationContext.xml");
9.     public static void main(String[] args) {
10.          Servlet servlet=(Servlet)context.getBean("servletref");
11.          servlet.servletMethod();
12.    }
13.  }
```

output

```
Servlet constructor
Servlet.servletMethod() called
Service.serviceMethod() called
```

Observe above application we provided both constructor and setter in Servlet.java but it injects by constructor approach because there is no default constructor.

Now provide default constructor also in Servlet.java and run the application.

Servlet.java

```
1. package com.neo.spring.bean;
2. public class Servlet {
3.     private Service service;
4.     public Servlet() {
5.           System.out.println("default constructor");
6.     }
7.     public Servlet(Service service) {
8.           this.service=service;
9.           System.out.println("Servlet constructor");
10.          }
11.          public void setService(Service service) {
12.                this.service = service;
13.                System.out.println("setter method");
14.          }
15.          public void servletMethod(){
16.                System.out.println("Servlet.servletMethod() called");
17.                service.serviceMethod();
18.          }
19.    }
```

output

```
setter method
Servlet.servletMethod() called
Service.serviceMethod() called
```

Now observe here injection is happened by using byType because default constructor is there.

Note: It's always good to combine both 'auto-wire' and 'dependency-check' together, to make sure the property is always auto-wire successfully.

```
<bean id="customer" class="com.neo.spring.bean.Customer"
                autowire="autodetect" dependency-check="objects />

<bean id="person" class="com.neo.spring.bean.Person" />
```

Conclusion

Spring 'auto-wiring' make development faster with great costs – it added complexity for the entire bean configuration file, and you don't even know which bean will auto wired in which bean.

In practice, i rather wire it manually, it is always clean and work perfectly, or better uses @Autowired annotation, which is more flexible and recommended.

Spring Auto-Wiring Beans with @Autowired annotation

In last Spring auto-wiring in XML example, it will autowired the matched property of any bean in current Spring container. In most cases, you may need autowired property in a particular bean only.

In Spring, you can use **@Autowired** annotation to auto wire bean on the setter method, constructor or a field. Moreover, it can autowired property in a particular bean.

Note
The @Autowired annotation is auto wire the bean by matching data type.

See following full example to demonstrate the use of **@Autowired**.

```
package com.neo.spring.bean;

public class Customer
{
	//you want autowired this field.
	private Person person;

	private int type;
	private String action;

	//getter and setter method

}
```

```
<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-2.5.xsd">

	<bean id="CustomerBean" class="com.neo.spring.bean.Customer">
		<property name="action" value="buy" />
		<property name="type" value="1" />
	</bean>

	<bean id="PersonBean" class="com.neo.spring.bean.Person">
		<property name="name" value="neo" />
		<property name="address" value="address 123" />
		<property name="age" value="28" />
	</bean>

</beans>
```

2. Register AutowiredAnnotationBeanPostProcessor

To enable **@Autowired**, you have to register '**AutowiredAnnotationBeanPostProcessor**', and you can do it in two ways :

1. Include <context:annotation-config />

Add Spring context and <context:annotation-config /> in bean configuration file.

```
<beans
	//...
	xmlns:context="http://www.springframework.org/schema/context"
	//...
	http://www.springframework.org/schema/context
	http://www.springframework.org/schema/context/spring-context-2.5.xsd">
```

```
        //...

        <context:annotation-config />
        //...
</beans>
```

Full example,

```
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:context="http://www.springframework.org/schema/context"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-2.5.xsd">

        <context:annotation-config />

        <bean id="CustomerBean" class="com.neo.spring.bean.Customer">
                <property name="action" value="buy" />
                <property name="type" value="1" />
        </bean>

        <bean id="PersonBean" class="com.neo.spring.bean.Person">
                <property name="name" value="neo" />
                <property name="address" value="address ABC" />
                <property name="age" value="29" />
        </bean>

</beans>
```

2. Include AutowiredAnnotationBeanPostProcessor

Include 'AutowiredAnnotationBeanPostProcessor' directly in bean configuration file.

```
<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-2.5.xsd">

<bean
class="org.springframework.spring.beans.factory.annotation.AutowiredAnnotationBeanPostProcesso
r"/>

        <bean id="CustomerBean" class="com.neo.spring.bean.Customer">
                <property name="action" value="buy" />
                <property name="type" value="1" />
        </bean>

        <bean id="PersonBean" class="com.neo.spring.bean.Person">
                <property name="name" value="neo" />
                <property name="address" value="address ABC" />
                <property name="age" value="29" />
        </bean>

</beans>
```

3. @Autowired Examples

Now, you can autowired bean via **@Autowired**, and it can be applied on setter method, constructor or a field.

1. @Autowired setter method

```
package com.neo.spring.bean;

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

public class Customer
{
        private Person person;
        private int type;
        private String action;
        //getter and setter methods

        @Autowired
        public void setPerson(Person person) {
                this.person = person;
        }
}
```

2. @Autowired construtor

```
package com.neo.spring.bean;

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

public class Customer
{
        private Person person;
        private int type;
        private String action;
        //getter and setter methods

        @Autowired
        public Customer(Person person) {
                this.person = person;
        }
}
```

3. @Autowired field

```
package com.neo.spring.bean;

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

public class Customer
{
        @Autowired
        private Person person;
        private int type;
        private String action;
        //getter and setter methods
}
```

The above example will autowired 'PersonBean' into Customer's person property.

Run it

```
package com.neo.spring.bean;
```

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class App
{
    public static void main( String[] args )
    {
        ApplicationContext context =
          new ClassPathXmlApplicationContext(new String[] {"SpringBeans.xml"});

        Customer cust = (Customer)context.getBean("CustomerBean");
        System.out.println(cust);

    }
}
```

Output

```
Customer [action=buy, type=1,
person=Person [address=address 123, age=28, name=neo]]
```

Dependency checking

By default, the @Autowired will perform the dependency checking to make sure the property has been wired properly. When Spring can't find a matching bean to wire, it will throw an exception. To fix it, you can disable this checking feature by setting the "**required**" attribute of @Autowired to false.

```
package com.neo.spring.bean;

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

public class Customer
{
        @Autowired(required=false)
        private Person person;
        private int type;
        private String action;
        //getter and setter methods
}
```

In the above example, if the Spring can't find a matching bean, it will leave the person property unset

@Qualifier

The @Qualifier annotation us used to control which bean should be autowire on a field. For example, bean configuration file with two similar person beans.

```
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:context="http://www.springframework.org/schema/context"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-2.5.xsd">

        <context:annotation-config />

        <bean id="CustomerBean" class="com.neo.spring.bean.Customer">
                <property name="action" value="buy" />
                <property name="type" value="1" />
        </bean>
```

```
        <bean id="PersonBean1" class="com.neo.spring.bean.Person">
                <property name="name" value="neo1" />
                <property name="address" value="address 1" />
                <property name="age" value="28" />
        </bean>

        <bean id="PersonBean2" class="com.neo.spring.bean.Person">
                <property name="name" value="neo2" />
                <property name="address" value="address 2" />
                <property name="age" value="28" />
        </bean>

</beans>
```

Will Spring know which bean should wire?

To fix it, you can use **@Qualifier** to auto wire a particular bean, for example,

```
package com.neo.spring.bean;

import org.springframework.spring.beans.factory.annotation.Autowired;
import org.springframework.spring.beans.factory.annotation.Qualifier;

public class Customer
{
        @Autowired
        @Qualifier("PersonBean1")
        private Person person;
        private int type;
        private String action;
        //getter and setter methods
}
```

It means, bean "PersonBean1" is autowired into the Customer's person property. Read this full example – Spring Autowiring @Qualifier example

Conclusion

This **@Autowired** annotation is highly flexible and powerful, and definitely better than "**autowire**" attribute in bean configuration file.

PropertyEditor

- PropertyEditor object is used to convert String representation of data into Object representation and vice versa.
- PropertyEditor is an interface that belongs to java bean API

Benefits of property editors in spring:
If we use property editors in spring, we get the flexibility in populating the bean fields directly with string type even though bean fields are of some object type.

Built-in spring framework property editors:
ByteArrayPropertyEditor, ClassEditor, CustomBooleanEditor, LocaleEditor, URLEditor, FileEditor, CustomCollectionEditor...etc like this we have so many built-in property editors are there. These all are used by spring framework internally.

Custom Property editor:
Custom Property editor means our own property editor class used to perform our own application specific conversations.

Steps to develop custom property editor:
Step 1: develop a spring bean (dependency bean) whose values are populated from String representation to object representation. In our project the dependency bean is **ContactNumber.java**

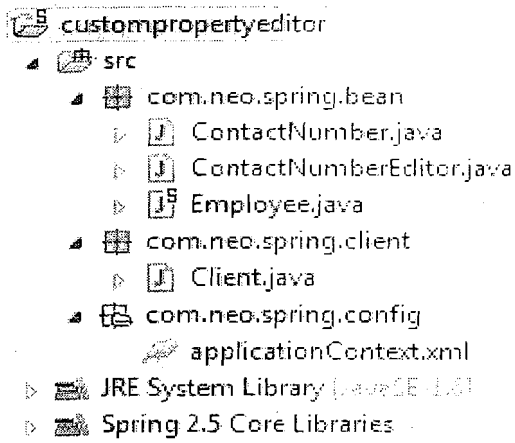
Step 2:
Develop the custom property editor class by extending **java.beans.PropertyEditorSupport** class. In our project the editor bean is **ContactNumberEditor.java**

Note:
Custom property editor class name should be suffixed with **Editor** to the dependency bean class and should be placed in the same package where dependency bean resides. If we do so we no need to perform 4th step.

Step3:
Override **setAsText(String value)** method in the custom property editor class and implement convertion code(String to required object). After conversion call **setValue(convertedObject)** method.

Step4:
Configure the custom property editor in the configuration file (applicationContext.xml) by configuring the class org.springframework.beans.factory.config.**CustomEditorConfigurer** with property name=**"customEditors".**

Project on spring Custom PropertyEditors:



ContactNumber.java

```
1. package com.neo.spring.bean;
2.
3. public class ContactNumber {
4.     private String countryCode;
5.     private String stdCode;
6.     private String actualNumber;
7.
8.     public ContactNumber(String countryCode, String stdCode, String
9.                                                          actualNumber) {
10.                this.countryCode = countryCode;
11.                this.stdCode = stdCode;
12.                this.actualNumber = actualNumber;
13.        }
14.
15.
16.    public String getContactNumber() {
17.                return countryCode + "-" + stdCode + "-" + actualNumber;
18.        }
19.    }
```

ContactNumberEditor.java

```
1. package com.neo.spring.bean;
2.
3. import java.beans.PropertyEditorSupport;
4.
5. public class ContactNumberEditor extends PropertyEditorSupport {
6.     @Override
7.     // container will pass injected contact number as a string
8.     public void setAsText(String text) throws IllegalArgumentException {
9.
10.                String[] token = text.split("-");
11.                ContactNumber contactNumber = new ContactNumber(token[0],
12.                                                token[1], token[2]);
13.
14.                setValue(contactNumber);
15.        }
```

```
16.   }
```

Employee.java

```
1. package com.neo.spring.bean;
2.
3. public class Employee {
4.     private int eno;
5.     private String ename;
6.     private String city;
7.     private ContactNumber contactNumber;
8.
9.     public void setEno(int eno) {
10.              this.eno = eno;
11.        }
12.
13.        public void setEname(String ename) {
14.              this.ename = ename;
15.        }
16.
17.        public void setCity(String city) {
18.              this.city = city;
19.        }
20.
21.        public void setContactNumber(ContactNumber contactNumber) {
22.              this.contactNumber = contactNumber;
23.        }
24.
25.        public void displayDetails() {
26.              System.out.println(eno);
27.              System.out.println(ename);
28.              System.out.println(city);
29.              System.out.println(contactNumber.getContactNumber());
30.        }
31.   }
```

applicationContext.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans
3.     xmlns="http://www.springframework.org/schema/beans"
4.     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5.     xmlns:p="http://www.springframework.org/schema/p"
6.     xsi:schemaLocation="http://www.springframework.org/schema/beans
7.     http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
8.
9. <bean id="emp" class="com.neo.spring.bean.Employee" >
10.         <property name="eno" value="1001" ></property>
11.         <property name="ename" value="sekhar" ></property>
12.         <property name="city" value="Hyderabad" ></property>
13.         <property name="contactNumber"  value="91-084984-232236" >
14.         </property>
15.   </bean>
16.
17.   <bean class="org.springframework.beans.factory.config
18.                                          .CustomEditorConfigurer" >
19.
```

```
20.   <property name="customEditors">
21.         <map>
22.               <entry key="com.neo.spring.bean.ContactNumber" >
23.               <bean class="com.neo.spring.bean.ContactNumberEditor"  >
24.                </bean>
25.               </entry>
26.         </map>
27.   </property>
28.
29.   </bean>
30.
31.   </beans>
```

Client.java

```
1. package com.neo.spring.client;
2. import org.springframework.context.ApplicationContext;
3. import org.springframework.context.support.ClassPathXmlApplicationContext;
4.
5. import com.neo.spring.bean.Employee;
6. public class Client {
7.    private static ApplicationContext context = new
8.                                  ClassPathXmlApplicationContext(
9.               "com/neo/spring/config/applicationContext.xml");
10.
11.         public static void main(String[] args) {
12.               Emplcyee employee = (Employee) context.getBean("emp");
13.               employee.displayDetails();
14.         }
15.   }
```

Ouput

```
1001
sekhar
Hyderabad
91-084984-232236
```

Internationalization(I18N)

- In software system Internationalization means based on the users country and language we need to display respective content in the browser.
- In JAVA in order to get the user country and language we can achieve by using **java.util.Locale** object.

LOCALE=language+country

A **java.util.Locale** is a lightweight object that contains only a few important members:

- A language code
- An optional country or region code
- An optional variant (in INDIA it is called as State) code

The notation of Locale:

< language code>[_<country code>[_<variant code>]]

In the above notation each code is separated with an underscore symbol (_).
The country code and variant codes are optional.

Language codes:

Language codes are defined by ISO 639 that assigns two- and three-letter codes to most languages of the world. Locale uses the two-letter codes to identify the target language.

Language Code Examples in the ISO 639 Standard

Language	Code
Arabic	Ar
German	De
English	En
Spanish	Es
Japanese	Ja
Hebrew	He

Country (Region) Codes:

Country codes are defined by ISO 3166, another international standard. It defines two- and three-letter abbreviations for each country or major region in the world. In contrast to the language codes, country codes are set uppercase.

Table 3. Some Country Codes Defined in the ISO 3166 Standard

Country	Code
United States	US
Canada	CA
France	FR
Japan	JP

Germany	DE

Variant (State) Code:

To provide additional functionality or customization we use Variant code, that isn't possible with just a language and country.

Ex: de_DE----------→ German-speaking German locale
de_DE_EURO----------→ German-speaking German locale with a euro variant
en_US_SiliconValley-----→ English speaking US locale for silicon valley people

In Java the syntax of above notation by three constructors:

- Locale (String language)
- Locale (String language, String country)
- Locale (String language, String country, String variant)

Example on the above notations:

```
// Creates a generic English-speaking locale.
 Locale locale1 = new Locale("en");
 // Creates an English-speaking, Canadian locale.
 Locale locale2 = new Locale("en", "CA");
 // Create a very specific English-speaking, U.S. locale for Silicon Valley.
 Locale locale3 = new Locale("en", "US", "SiliconValley");
```

STEPS to write I18N application in spring:

In three steps we can create I18N application.

Step1:

Create the required properties files.

Syntax for naming the properties file:

We can write this in three ways.

1. Propertiesfilename_languagecode.properties
2. Propertiesfilename_languagecode_countrycode.properties
3. Propertiesfilename_languagecode_countrycode_varientcode.properties

Our project property files:

In this we have created four properties files. The properties file should be in the form of key and value pair. In our properties files the key is '**welcome. message**'.

1.ApplicationResources_en_US_SiliconValley.properties

welcome.message=welcome to US silican vally english users {0}.{1}

note:

{0},{1}→used to pass some values dynamically, called as place holders

2.ApplicationResources_fr_CA.properties

welcome.message=welcome to french CANADA english users

3.ApplicationResources_zh_CN.properties

welcome.message=welcome to chineese english users

4.ApplicationResources.properties

welcome.message=welcome to users

Note: in the above properties file we didn't mention any language code and country code this will execute by default if it doesn't find any codes.

Step 2:

Configure the properties files in the configuration file.

- Inject the properties file prefix to spring provided bean "**ResourceBundleMessageSource**" using **'basename' or 'basenames'** values for the **name** attribute of **<bean>** tag and also the id reference for this bean should be **'messageSource'**.
- If we are injecting one properties file then we can use 'basename 'or 'basenames' value for the name attribute **without <list> tag**.
- In case if we are injecting more than one properties file then we should use only 'basenames' value for the name attribute **with <list> tag**.

Client program:

Here to get the language content we should call getMessage(key, arguments, locale) method of ApplicationContext of by passing

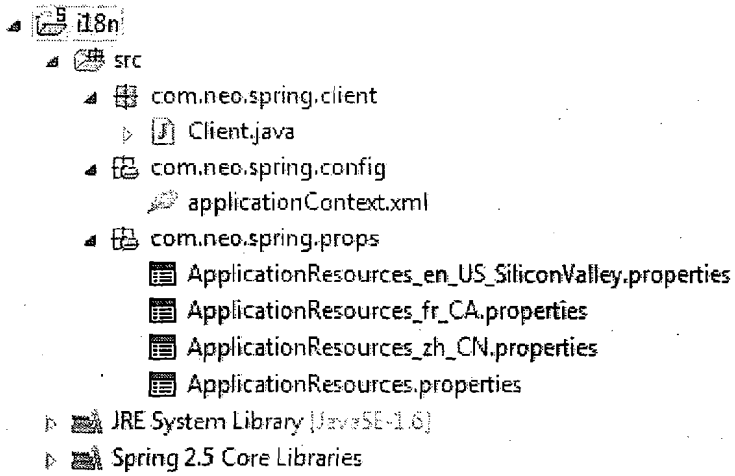
1. Key
2. Arguments to the message
3. Locale Object

In Our project :

Key----------→ **welcome.message**

Messege arguments---→dynamic values for the place holders i.e., **{0}{1}**

Note:if we don't have any arguments then we should pass **null** as an argument



Client.java:

```
1. package com.neo.spring.client;
2. import java.util.Locale;
3. import org.springframework.context.ApplicationContext;
4. import org.springframework.context.support.ClassPathXmlApplicationContext;
5. public class Client {
6.     private static ApplicationContext context = new
7.                     ClassPathXmlApplicationContext(
```

```
8.                "com/neo/spring/config/applicationContext.xml");
9.        public static void main(String[] args) {
10.               String message = context.getMessage("welcome.message", new
11.                     Object[] { "US", "English" }, new Locale("en", "US",
12.                                                        "SiliconValley"));
13.               System.out.println(message);
14.              message = context.getMessage("welcome.message", null, new
15.                                                      Locale("zh","CN"));
16.               System.out.println(message);
17.                              message = context.getMessage(
18.                              "welcome.message", null, new Locale("Ar"));
19.               System.out.println(message);
20.        }
21.  }
```

applicationContext.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3.     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.     xmlns:p="http://www.springframework.org/schema/p"
5.     xsi:schemaLocation="http://www.springframework.org/schema/beans
6.     http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
7.     <bean id="messageSource" class="org.springframework.context.support
8.                                          .ResourceBundleMessageSource">
9.          <property name="basename"
10.                      value="com/neo/spring/props/ApplicationResources">
11.                </property>
12.         </bean>
13.       </beans>
```

ApplicationResources_en_US_SiliconValley.properties

```
1. Welcome.message=This is the {0}{1} welcome message
```

ApplicationResources_fr_CA.properties:

```
1. Welcome.message=This is the Canadian French welcome message
```

ApplicationResources_zh_CN.properties:

```
1. Welcome.message=This is the Chinese welcome message
```

ApplicationResources. Properties:

```
1. Welcome.message=This is the default message
```

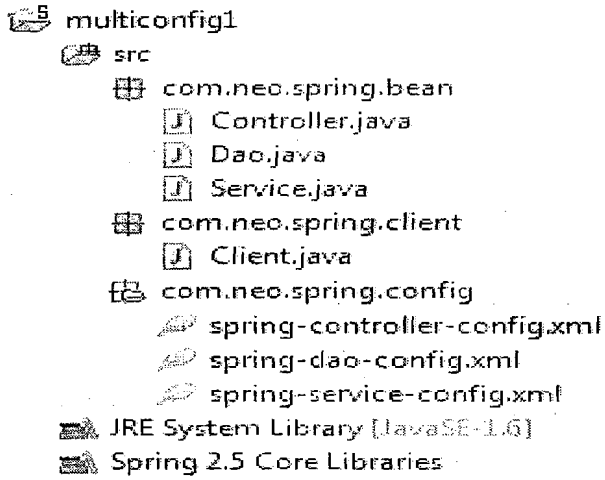
Output

```
This is the US English welcome message
This is the chinees welcome message
This is the default message
```

Q.)How to work with multiple spring configuration files?

- When the spring configuration file size is huge, it is difficult to maintain, so in that scenarios we generally have multiple configuration files.
- We can work with multiple spring configuration files in two ways
  - By importing other spring file into the current configuration file.
    - Ex: **<import resource="other-spring-beans.xml"/>**
  - By passing all spring bean configuration files as a string array to the constructor of the container
    - new ClassPathXmlApplicationContext(new String[] {"beans-1.xml", "beans-2.xml.xml","beans-3.xml"});

Multiconfig1



Controller.java

```
1. package com.neo.spring.bean;
2.
3. public class Controller {
4.     private Service service;
5.
6.     public void setService(Service service) {
7.           this.service = service;
8.     }
9.
10.          public void controllerMethod() {
11.                System.out.println("Controller.controllerMethod()");
12.                service.serviceMethod();
13.          }
14.    }
```

Dao.java

```
1. package com.neo.spring.bean;
2.
3. public class Dao {
4.     public void daoMethod() {
5.           System.out.println("Dao.daoMethod()");
6.     }
7. }
```

Service.java

```
1. package com.neo.spring.bean;
2.
3. public class Service {
4.     private Dao dao;
5.
6.     public void setDao(Dao dao) {
7.           this.dao = dao;
8.     }
9.
10.          public void serviceMethod() {
11.                System.out.println("Service.serviceMethod()");
12.                dao.daoMethod();
13.          }
14.    }
```

Client.java

```
1. package com.neo.spring.client;
2.
3. import org.springframework.context.ApplicationContext;
4. import org.springframework.context.support.ClassPathXmlApplicationContext;
5.
6. import com.neo.spring.bean.Controller;
7.
8. public class Client {
9.     private static ApplicationContext context = new
10.                        ClassPathXmlApplicationContext(
11.                "com/neo/spring/config/spring-controller-config.xml");
12.
13.          public static vod main(String[] args) {
14.                Controller controller = (Controller)
15.                                        context.getBean("controllerref");
16.                controller.controllerMethod();
17.          }
18.    }
```

spring-controller-config.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3.     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.     xmlns:p="http://www.springframework.org/schema/p"
5.     xsi:schemaLocation="http://www.springframework.org/schema/beans
6.     http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
7.     <import resource="spring-service-config.xml" />
8.     <bean id="controllerref" class="com.neo.spring.bean.Controller">
9.           <property name="service" ref="serviceref" ></property>
10.          </bean>
11.    </beans>
```

spring-dao-config.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3.     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.     xmlns:p="http://www.springframework.org/schema/p"
5.     xsi:schemaLocation="http://www.springframework.org/schema/beans
```

```
6.    http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
7.    <bean id="daoref" class="com.neo.spring.bean.Dao">
8.    </bean>
9. </beans>
```

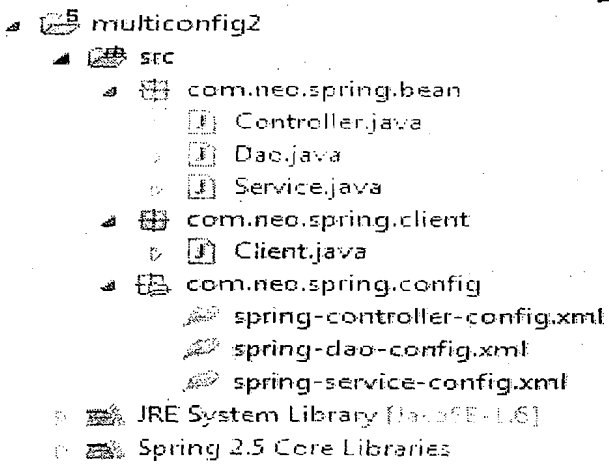
spring-service-config.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3.    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.    xmlns:p="http://www.springframework.org/schema/p"
5.    xsi:schemaLocation="http://www.springframework.org/schema/beans
6.    http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
7.
8.    <import resource="spring-dao-config.xml" />
9.    <bean id="serviceref" class="com.neo.spring.bean.Service">
10.         <property name="dao" ref="daoref" ></property>
11.         </bean>
12.   </beans>
```

Output

```
Controller.controllerMethod()
Service.serviceMethod()
Dao.daoMethod()
```

Multiconfig2



Controller.java

```
1. package com.neo.spring.bean;
2.
3. public class Controller {
4.    private Service service;
5.
6.    public void setService(Service service) {
7.          this.service = service;
8.    }
9.
10.         public void controllerMethod() {
11.               System.out.println("Controller.controllerMethod()");
12.               service.serviceMethod();
13.         }
```

```
14.   }
```

Dao.java

```
1. package com.neo.spring.bean;
2.
3. public class Dao {
4.     public void daoMethod() {
5.           System.out.println("Dao.daoMethod()");
6.     }
7. }
```

Service.java

```
1. package com.neo.spring.bean;
2.
3. public class Service {
4.     private Dao dao;
5.
6.     public void setDao(Dao dao) {
7.           this.dao = dao;
8.     }
9.
10.          public void serviceMethod() {
11.                System.out.println("Service.serviceMethod()");
12.                dao.daoMethod();
13.          }
14.    }
```

Client.java

```
1. package com.neo.spring.client;
2.
3. import org.springframework.context.ApplicationContext;
4. import org.springframework.context.support.ClassPathXmlApplicationContext;
5.
6. import com.neo.spring.bean.Controller;
7.
8. public class Client {
9.     private static String[] configs = {
10.                    "com/neo/spring/config/spring-controller-config.xml",
11.                    "com/neo/spring/config/spring-service-config.xml",
12.                    "com/neo/spring/config/spring-dao-config.xml" };
13.          private static ApplicationContext context = new
14.                                   ClassPathXmlApplicationContext(configs);
15.
16.          public static void main(String[] args) {
17.                Controller controller = (Controller)
18.                                              context.getBean("controllerref");
19.                controller.controllerMethod();
20.          }
21.    }
```

spring-controller-config.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3.     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.     xmlns:p="http://www.springframework.org/schema/p"
```

```
5.     xsi:schemaLocation="http://www.springframework.org/schema/beans
6.     http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
7.
8.     <bean id="controllerref" class="com.neo.spring.bean.Controller">
9.           <property name="service" ref="serviceref" ></property>
10.          </bean>
11.    </beans>
```

spring-dao-config.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3.     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.     xmlns:p="http://www.springframework.org/schema/p"
5.     xsi:schemaLocation="http://www.springframework.org/schema/beans
6.     http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
7.
8.     <bean id="daoref" class="com.neo.spring.bean.Dao">
9.     </bean>
10.    </beans>
```

spring-service-config.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3.     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.     xmlns:p="http://www.springframework.org/schema/p"
5.     xsi:schemaLocation="http://www.springframework.org/schema/beans
6.     http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
7.     <bean id="serviceref" class="com.neo.spring.bean.Service">
8.     <property name="dao" ref="daoref" ></property>
9.     </bean>
10.    </beans>
```

Output

```
Controller.controllerMethod()
Service.serviceMethod()
Dao.daoMethod()
```

References to other beans (collaborators)

The `ref` element is the final element allowed inside a `<constructor-arg/>` or `<property/>` definition element. It is used to set the value of the specified property to be a reference to another bean managed by the container (a collaborator). As mentioned in a previous section, the referred-to bean is considered to be a dependency of the bean who's property is being set, and will be initialized on demand as needed (if it is a singleton bean it may have already been initialized by the container) before the property is set. All references are ultimately just a reference to another object, but there are 3 variations on how the id/name of the other object may be specified, which determines how scoping and validation is handled.

Specifying the target bean by using the `bean` attribute of the `<ref/>` tag is the most general form, and will allow creating a reference to any bean in the same container (whether or not in the same XML file), or parent container. The value of the `'bean'` attribute may be the same as

either the 'id' attribute of the target bean, or one of the values in the 'name' attribute of the target bean.

```
<ref bean="someBean"/>
```

Specifying the target bean by using the local attribute leverages the ability of the XML parser to validate XML id references within the same file. The value of the local attribute must be the same as the id attribute of the target bean. The XML parser will issue an error if no matching element is found in the same file. As such, using the local variant is the best choice (in order to know about errors as early as possible) if the target bean is in the same XML file.

```
<ref local="someBean"/>
```

Specifying the target bean by using the 'parent' attribute allows a reference to be created to a bean which is in a parent container of the current container. The value of the 'parent' attribute may be the same as either the 'id' attribute of the target bean, or one of the values in the 'name' attribute of the target bean, and the target bean must be in a parent container to the current one. The main use of this bean reference variant is when you have a hierarchy of containers and you want to wrap an existing bean in a parent container with some sort of proxy which will have the same name as the parent bean.

```
<!-- in the parent context -->
<bean id="accountService" class="com.foo.SimpleAccountService">
    <!-- insert dependencies as required as here -->
</bean>
<!-- in the child (descendant) context -->
<bean id="accountService"  <-- notice that the name of this bean is the same as
the name of the 'parent' bean
      class="org.springframework.aop.framework.ProxyFactoryBean">
      <property name="target">
          <ref parent="accountService"/>  <-- notice how we refer to the parent
bean
      </property>
    <!-- insert other configuration and dependencies as required as here -->
</bean>
```

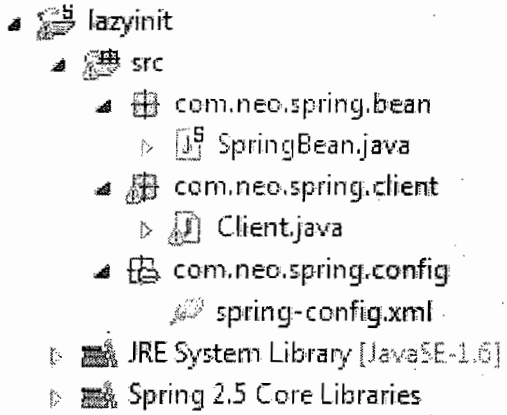
Q.) What is lazy-init attribute of <bean> tag?

=> When we are using ApplicationContext based container, As and when we creates the spring container, it creates all the beans (Having scope value as '**singleton**'- we know that default scope is '**singleton**') which are configured in the spring configuration file.

=> Sometimes we don't want to create the beans until and unless the request comes**(call getBean("bean-ref"))** for particular bean. This can be achieved with **lazy-init** attribute of **<bean>** tag.

=> if we configure, **lazy-init="true"** for any bean, that bean object will not be created when we create the ApplicationContext based container. That will be created only when we call **getBean("bean-ref")**.

Lazyinit



SpringBean

```
1. package com.neo.spring.bean;
2.
3. public class SpringBean {
4.     public SpringBean() {
5.           System.out.println("SpringBean is created");
6.     }
7.
8. }
```

Client.java

```
1. package com.neo.spring.client;
2.
3. import org.springframework.context.ApplicationContext;
4. import org.springframework.context.support.ClassPathXmlApplicationContext;
5.
6. public class Client {
7.     private static ApplicationContext context = new
8.                                   ClassPathXmlApplicationContext(
9.                "com/neo/spring/config/spring-config.xml");
10.
11.          public static void main(String[] args) {
12.                System.out.println("testing");
13.          }
14.    }
```

spring-config.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
```

```
2. <beans xmlns="http://www.springframework.org/schema/beans"
3.     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.     xmlns:p="http://www.springframework.org/schema/p"
5.     xsi:schemaLocation="http://www.springframework.org/schema/beans
6.     http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
7.     <bean id="springBean" lazy-init="true"
8.                           class="com.neo.spring.bean.SpringBean"></bean>
9.
10.    </beans>
```

Output

```
testing
```

NOTE: Here spring container is not creating "springBean" object, as and when container is started. Now remove lazy-init attribute, then the output is as follows.

spring-config.xml

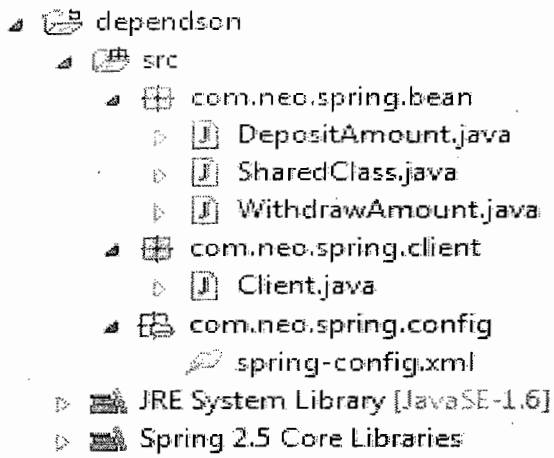
```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3.     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.     xmlns:p="http://www.springframework.org/schema/p"
5.     xsi:schemaLocation="http://www.springframework.org/schema/beans
6.     http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
7.     <bean id="springBean"
8.                           class="com.neo.spring.bean.SpringBean"></bean>
9.
10.    </beans>
```

Output

```
testing
SpringBean is created
```

Q.) What is depends-on attribute of <bean> tag?
=> sometimes some bean creation is meaningful only when its dependency is created.
=> For example, Vehicle bean creation is meaningful, when the Petrol bean is available.

Depends-on



DepositAmount.java

```
1. package com.neo.spring.bean;
2.
3. public class DepositAmount {
4.
5.     /** Creates a new instance of DepositAmount */
6.     public DepositAmount() {
7.           System.out.println("Before Depositing : " +
8.                                                   SharedClass.getAmount());
9.           SharedClass.setAmount(SharedClass.getAmount() + 100);
10.                System.out.println("After Depositing : " +
11.                SharedClass.getAmount());
12.          }
13.    }
```

SharedClass.java

```
1. package com.neo.spring.bean;
2. public class SharedClass {
3.      private static int amount;
4.      public static int getAmount() {
5.          return amount;
6.      }
7.
8.      public static void setAmount(int aAmount) {
9.          amount = aAmount;
10.        }
11.    }
```

WithdrawAmount.java

```
1. package com.neo.spring.bean;
2.
3. public class WithdrawAmount {
4.
5.     /** Creates a new instance of WithdrawAmount */
6.     public WithdrawAmount() {
```

```
7.          System.out.println("Before Withdrawal : " +
8.                                  SharedClass.getAmount());
9.          SharedClass.setAmount(SharedClass.getAmount() - 100);
10.               System.out.println("After Withdrawal :" +
11.                                 SharedClass.getAmount());
12.         }
13.   }
```

Client.java

```
1. package com.neo.spring.client;
2.
3. import org.springframework.context.ApplicationContext;
4. import
5.  org.springframework.context.support.ClassPathXmlApplicationContext;
6.
7. public class Client {
8.    private static ApplicationContext context = new
9.                      ClassPathXmlApplicationContext(
10.                    "com/neo/spring/config/spring-config.xml");
11.
12.         public static void main(String[] args) {
13.               context.getBean("withdraw");
14.               context.getBean("deposit");
15.
16.         }
17.   }
```

spring-config.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3.    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.    xmlns:p="http://www.springframework.org/schema/p"
5.    xsi:schemaLocation="http://www.springframework.org/schema/beans
6.    http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
7.    <bean id="sharedBean" class="com.neo.spring.bean.SharedClass" />
8.
9.    <!--  Execute Client.java without depends-on attribute -->
10.         <bean id="withdraw" class="com.neo.spring.bean.WithdrawAmount"
11.               dependes-on="deposit" ></bean>
12.         <bean id="deposit" class="com.neo.spring.bean.DepositAmount">
13.
14.         </bean>
15.   </beans>
```

With depends-on="true" output

```
Before Depositing : 0
After Depositing : 100
Before Withdrawal : 100
After Withdrawal :0
```

Without depends-on="true" Output

```
Before Withdrawal : 0
After Withdrawal :-100
Before Depositing : -100
After Depositing : 0
```

Q.) What is method-injection?

=> If dependent, dependency both are singleton **(scope='singleton')** objects, then there is no problem in the injection of dependency into dependent.

=> If dependent, dependency both are prototype **(scope='prototype')** objects, then there is no problem in the injection of dependency into dependent.

=> If dependent is prototype **(scope='prototype')** object, dependency is singleton **(scope='singleton')** objects, then there is no problem in the injection of dependency into dependent.

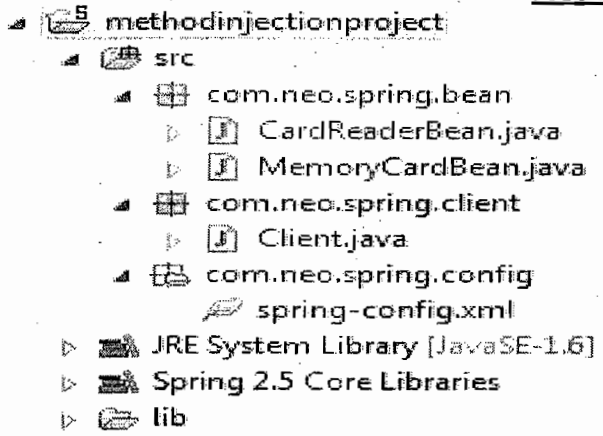
=> If dependent is singleton **(scope='singleton')** object, dependency is prototype **(scope='prototype')** objects, then there is problem in the injection of dependency into dependent.

=> Because dependent object created only once, but dependent object we are expecting as prototype. But as because of dependent object creation happens only once, then there is no chance of creating multiple dependency objects, and inject them into singleton dependent object.

=> To solve this problem, spring has given feature called **Method injection.**

=> Lookup method injection refers to the ability of the container to override methods on *container managed beans*, to return the result of looking up another named bean in the container. The lookup will typically be of a prototype bean as in the scenario described above. The Spring Framework implements this method injection by dynamically generating a subclass overriding the method, using bytecode generation via the CGLIB library.

MethodInjectionProject



CardReaderBean.java

```
1. package com.neo.spring.bean;
2.
3. public abstract class CardReaderBean {
4.
5.     public abstract MemoryCardBean getMemoryCardBean();
6.
7.     public void readDataFromCard() {
8.           System.out.println(getMemoryCardBean());
9.           System.out.println("CardReaderBean.reading data from card....");
10.                getMemoryCardBean().provideData();
11.          }
12.    }
```

MemoryCardBean.java

```
1. package com.neo.spring.bean;
2.
3. public class MemoryCardBean {
4.
5.    public void  provideData(){
6.          System.out.println("MemoryCardBean.providing data to card
7.                                                          reader.....");
8.    }
9. }
```

Client.java

```
1. package com.neo.spring.client;
2.
3. import org.springframework.context.ApplicationContext;
4. import org.springframework.context.support.ClassPathXmlApplicationContext;
5.
6. import com.neo.spring.bean.CardReaderBean;
7.
8. public class Client {
9.    private static ApplicationContext context = new
10.                      ClassPathXmlApplicationContext(
11.                     "com/neo/spring/config/spring-config.xml");
12.
13.         public static void main(String[] args) {
14.               System.out.println("first time calling method on card
15.                                                          reader");
16.               CardReaderBean readerBean = (CardReaderBean) context
17.                         .getBean("cardReaderBean");
18.               readerBean.readDataFromCard();
19.
20.               System.out.println("second time calling method on card
21.                                                          reader");
22.               readerBean = (CardReaderBean)
23.                                    context.getBean("cardReaderBean");
24.               readerBean.readDataFromCard();
25.
26.               System.out.println("third time calling method on card
27.                                                          reader");
28.               readerBean = (CardReaderBean)
29.                                    context.getBean("cardReaderBean");
30.               readerBean.readDataFromCard();
31.         }
32.   }
```

spring-config.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3.    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.    xmlns:p="http://www.springframework.org/schema/p"
5.    xsi:schemaLocation="http://www.springframework.org/schema/beans
6.    http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
7.    <bean id="memoryCardBean" class="com.neo.spring.bean.MemoryCardBean"
8.          scope="prototype"></bean>
9.    <bean id="cardReaderBean" class="com.neo.spring.bean.CardReaderBean"
```

```
10.                                                    scope="singleton" >
11.              <lookup-method name="getMemoryCardBean"
12.                             bean="memoryCardBean" />
13.         </bean>
14.
15.    </beans>
```

Output

```
first time calling method on card reader
com.neo.spring.bean.MemoryCardBean@1878144
CardReaderBean.reading data from card....
MemoryCardBean.providing data to card reader.....
second time calling method on card reader
com.neo.spring.bean.MemoryCardBean@137d090
CardReaderBean.reading data from card....
MemoryCardBean.providing data to card reader.....
third time calling method on card reader
com.neo.spring.bean.MemoryCardBean@15db314
CardReaderBean.reading data from card....
MemoryCardBean.providing data to card reader.....
```

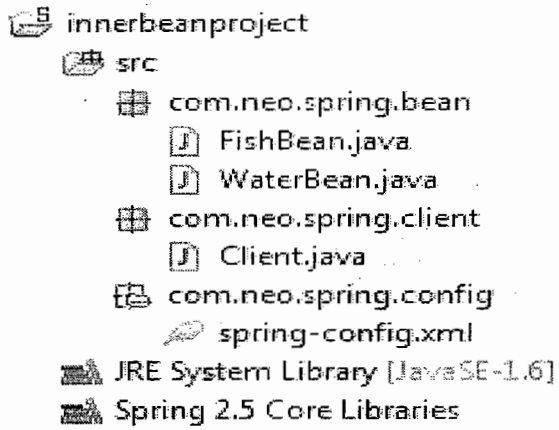
Q.) What is an inner bean configuration?

=> A spring bean configured with in <property> or <constructor-arg> tag using <bean> tag is known as inner bean configuration.

=> While configuring inner bean name/id attributes are not useful. So we say inner bean also called anonymous bean.

Example:

Innerbeanproject



FishBean.java

```
1. package com.neo.spring.bean;
2.
3. public class FishBean {
4.     public void swim(){
5.           System.out.println("Fishes are swimming ~:>, ~:>, ~:>");
6.     }
7. }
```

WaterBean.java

```
1. package com.neo.spring.bean;
2.
3. public class WaterBean {
4.     private FishBean fishBean;
5.
6.     public void setFishBean(FishBean fishBean) {
7.           this.fishBean = fishBean;
8.     }
9.
10.          public void flowing() {
11.                System.out.println("Water is flowing ~~~~~~~~~~~~");
12.                fishBean.swim();
13.                System.out.println("Water is flowing ~~~~~~~~~~~~");
14.          }
15.    }
```

Client.java

```
1. package com.neo.spring.client;
2.
3. import org.springframework.context.ApplicationContext;
4. import org.springframework.context.support.ClassPathXmlApplicationContext;
```

```
5.
6. import com.neo.spring.bean.WaterBean;
7.
8. public class Client {
9.     private static ApplicationContext context = new
10.                         ClassPathXmlApplicationContext(
11.                        "com/neo/spring/config/spring-config.xml");
12.
13.         public static void main(String[] args) {
14.               WaterBean waterBean = (WaterBean)
15.                                       context.getBean("waterBean");
16.               waterBean.flowing();
17.         }
18.   }
```

spring-config.xml

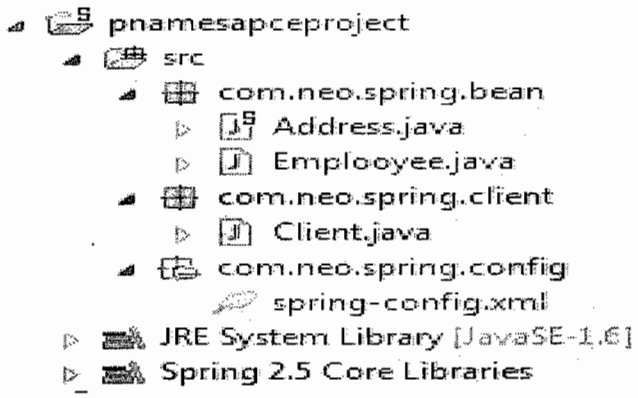
```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3.     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.     xmlns:p="http://www.springframework.org/schema/p"
5.     xsi:schemaLocation="http://www.springframework.org/schema/beans
6.     http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
7.
8.     <bean id="waterBean" class="com.neo.spring.bean.WaterBean">
9.           <property name="fishBean">
10.                      <bean class="com.neo.spring.bean.FishBean"></bean>
11.               </property>
12.          </bean>
13.    </beans>
```

Output

```
Water is flowing ~~~~~~~~~~~~
Fishes are swimming ~:>, ~:>, ~:>
Water is flowing ~~~~~~~~~~~~
```

Q.) What is p-namespace?

- ⇨ use the special "p-namespace" To limit the amount of XML you have to write, to configure your beans.
- ⇨ Instead of using **nested** property elements, using the p-namespace you can use attributes as part of the bean element that describe your property values. The values of the attributes will be taken as the values for your properties.
- ⇨ P-namespace is used for setter injection



Address.java

```
1. package com.neo.spring.bean;
2.
3. public class Address {
4.     private String hno;
5.     private String city;
6.
7.     public String getHno() {
8.           return hno;
9.     }
10.
11.          public void setHno(String hno) {
12.                this.hno = hno;
13.          }
14.
15.          public String getCity() {
16.                return city;
17.          }
18.
19.          public void setCity(String city) {
20.                this.city = city;
21.          }
22.    }
```

Emplooyee.java

```
1. package com.neo.spring.bean;
2.
3. public class Emplooyee {
4.     private int eno;
5.     private String ename;
6.     private Address address;
7.
8.     public void setEno(int eno) {
```

```
9.          this.eno = eno;
10.         }
11.
12.         public void setEname(String ename) {
13.               this.ename = ename;
14.         }
15.
16.         public void setAddress(Address address) {
17.               this.address = address;
18.         }
19.
20.         public void displayDetails() {
21.               System.out.println("Employee details are....");
22.               System.out.println(eno);
23.               System.out.println(ename);
24.               System.out.println(address.getHno());
25.               System.out.println(address.getCity());
26.         }
27.
28.   }
```

Client.java

```
1. package com.neo.spring.client;
2.
3. import org.springframework.context.ApplicationContext;
4. import org.springframework.context.support.ClassPathXmlApplicationContext;
5.
6. import com.neo.spring.bean.Emplooyee;
7.
8. public class Client {
9.    private static ApplicationContext context = new
10.                     ClassPathXmlApplicationContext(
11.                   "com/neo/spring/config/spring-config.xml");
12.
13.         public static void main(String[] args) {
14.               Emplooyee emplooyee = (Emplooyee)
15.                                       context.getBean("employee");
16.               emplooyee.displayDetails();
17.         }
18.   }
```

spring-config.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3.    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.    xmlns:p="http://www.springframework.org/schema/p"
5.    xsi:schemaLocation="http://www.springframework.org/schema/beans
6.    http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
7.    <!--
8.          <bean id="address" class="com.neo.spring.bean.Address">
9.          <property name="hno"> <value>152-A</value> </property>
10.         <property name="city">
11.               <value>Hyderabad</value> </property>
12.       </bean>
13.   <bean id="employee" class="com.neo.spring.bean.Emplooyee">
```

```
14.       <property name="eno">
15.               <value>1002</value> </property> <property name="ename">
16.               <value>sekhar</value> </property> <property name="address">
17.               <ref bean="address" /> </property> </bean>
18.         -->
19.         <!--
20.               <bean id="address" class="com.neo.spring.bean.Address">
21.               <propertyname="hno" value="152-A" />
22.               <property name="city" value="Hyderabad" />
23.               </bean>
24.    <bean id="employee" class="com.neo.spring.bean.Emplooyee">
25.               <property name="eno" value="1002" />
26.               <property name="address" ref="address" /> </bean>
27.         -->
28.
29.         <bean id="address" class="com.neo.spring.bean.Address"
30.                              p:hno="152-A" p:city="Hyderabad" />
31.         <bean id="employee" class="com.neo.spring.bean.Emplooyee"
32.               p:eno="1002" p:ename="sekhar" p:address-ref="address" />
33.
34.   </beans>
```

Output

```
Employee details are....
1002
sekhar
152-A
Hyderabad
```

Excluding a bean from being available for autowiring

You can also (on a per-bean basis) totally exclude a bean from being an autowire candidate. When configuring beans using Spring's XML format, the `'autowire-candidate'` attribute of the `<bean/>` element can be set to `'false'`; this has the effect of making the container totally exclude that specific bean definition from being available to the autowiring infrastructure.

This can be useful when you have a bean that you absolutely never ever want to have injected into other beans via autowiring. It does not mean that the excluded bean cannot itself be configured using autowiring... it can, it is rather that it itself will not be considered as a candidate for autowiring other beans.

Nulls

The `<null/>` element is used to handle `null` values. Spring treats empty arguments for properties and the like as empty `Strings`. The following XML-based configuration metadata snippet results in the email property being set to the empty `String` value ("")

```
<bean class="ExampleBean">
  <property name="email"><value/></property>
</bean>
```

This is equivalent to the following Java code: `exampleBean.setEmail("")`. The special `<null>` element may be used to indicate a `null` value. For example:

```
<bean class="ExampleBean">
  <property name="email"><null/></property>
</bean>
```

The above configuration is equivalent to the following Java code: `exampleBean.setEmail(null)`.

Strongly-typed collection (Java 5+ only)

If you are using Java 5 or Java 6, you will be aware that it is possible to have strongly typed collections (using generic types). That is, it is possible to declare a `Collection` type such that it can only contain `String` elements (for example). If you are using Spring to dependency inject a strongly-typed `Collection` into a bean, you can take advantage of Spring's type-conversion support such that the elements of your strongly-typed `Collection` instances will be converted to the appropriate type prior to being added to the`Collection`.

```
public class Foo {

    private Map<String, Float> accounts;

    public void setAccounts(Map<String, Float> accounts) {
        this.accounts = accounts;
    }
}
<beans>
    <bean id="foo" class="x.y.Foo">
        <property name="accounts">
            <map>
                <entry key="one" value="9.99"/>
                <entry key="two" value="2.75"/>
                <entry key="six" value="3.99"/>
            </map>
        </property>
    </bean>
</beans>
```

When the `'accounts'` property of the `'foo'` bean is being prepared for injection, the generics information about the element type of the strongly-typed `Map<String, Float>` is actually available via reflection, and so Spring's type conversion infrastructure will actually recognize the various value elements as being of type `Float` and so the string values `'9.99'`, `'2.75'`, and `'3.99'` will be converted into an actual `Float` type.

SPRING DAO / SPRING JDBC

Spring JDBC/DAO is one of the modules of spring. This will be used to build Persistent tier of Enterprise application (to develop DAO classes). This module is used to access the data from the Database (to do database operations).

DAO:

It is the Design pattern which seperates **Data Access Logic** from **Business Logic**.DAO classes can be developed with JDBC API, Hibernate, Ibatis, JPO, **Spring/JDBC** etc... If we use JDBC API explicitly in DAO classes development, there are some common practices (**boilerplate code**) has to implement in all DAO methods.

Example for common code or BOILER PLATE code:

```
class MyDAO{
public void myDaoMethod001(){

        String query = "SELECT SNO, SNAME, SAGE FROM STUDNT  WHERE SNO=?, ..... ";
   try{
        Connection connection= DriverManger.getConnection("url","username","password");
        Connection.setAutoCommit(false);
        PreparedStatement ps=connection.prepareStatement();
        ps.setInt(1, ....);
        ps.set...(2, .....)
        .....
        Resultset rs=stmt.executeQuery(query);
        while (rs.next()){
                int x=rs.getInt(1);
                String y=rs.getString(2);
                .....
        }
        con.commt();
    }catch(Exception e){
        ....
        con.rollback();
    }
    finally {
        rs.close();
        ps.close();
        con.close();
   }
}//my dao method001()

public void myDaoMethod002{
```

```
        .....
        .....
        con.setAutoCommit(false);
        try{
                Connection con=.....
                PreparedStatement ps=.....
                ResultSet rs=.....
                .....
                .....

                con.commit();
        }catch(Exception e){

                con.rollback();
                e.printstackTrace();
        }
        finally{
        con.close();
        ps.close();
        rs.close();
}// myDaoMethod()

}//MyDAO class
```

Note: All the above blocks are...Common Logics, if we use JDBC in our application.

As we can see in the above application there are some of the following common practices are there in all DAO methods.

1. Exception Handling Logic
2. Transaction Management Logic
3. Resource allocation Logic
4. Resource Releasing logic

Exception Handling logic

In JDBC

If we use JDBC API directly in DAO methods for all abnormal situations it will through **Java.sql.SqlException**..

Based on the error code available in the SQL Exception the developers needs to give the proper information to the client. This Exception Handling Logic we are explicitly writing in the source code of the application(If we need any modification to the Exception Handling Logic , we have to change the source code).

Java.sql.SqlException checked excption

In Spring:

Spring Frame Work provides the **Fine Grind Exception Handling** mechanism to deal with DataBase i.e. it defines specific Exceptions for each and every problem that occurs while dealing with the DB.

DataAccessException: org.springframework.dao.DataAccessException is the **top most Exception** in spring DAO exception hierarchy.

In Spring DAO Exception Hierarchy we have a support (or **Spring ExceptionTranslator)** to Transform lo-level Data Access API Exceptions to the Spring DAO Exceptions.

DataAccessException is the child class of **RuntimeException** so all Spring DAO Exceptions are unchecked Exceptions

Spring provides **Declarative Exception Handling** Mechanism, means we can handle the Exceptions in the **"Spring–Configuration"** file.

Transaction Logic:

In JDBC

In JDBC API we can implement the transactions using...
connection.setAutoCommit(boolean b);
connection.commit();
connection.rollback();

Using JDBC API we are implementing the transactions programmatically.

JDBC API doesn't support **Distributive Transaction Management.**

In spring:
Using spring we can implement transaction management declaratively.
Spring Framework supports Distributive Transaction Management.

Resource Allocation Logic:

In JDBC

In JDBC API DataBase resources i.e connection, statement, preparedStatement, callbackStatement....etc has to be allocated (created) explicitly by the developer

In Spring:

JdbcTemplate provided by the spring which, abstract the DB Resource Allocation Logic.

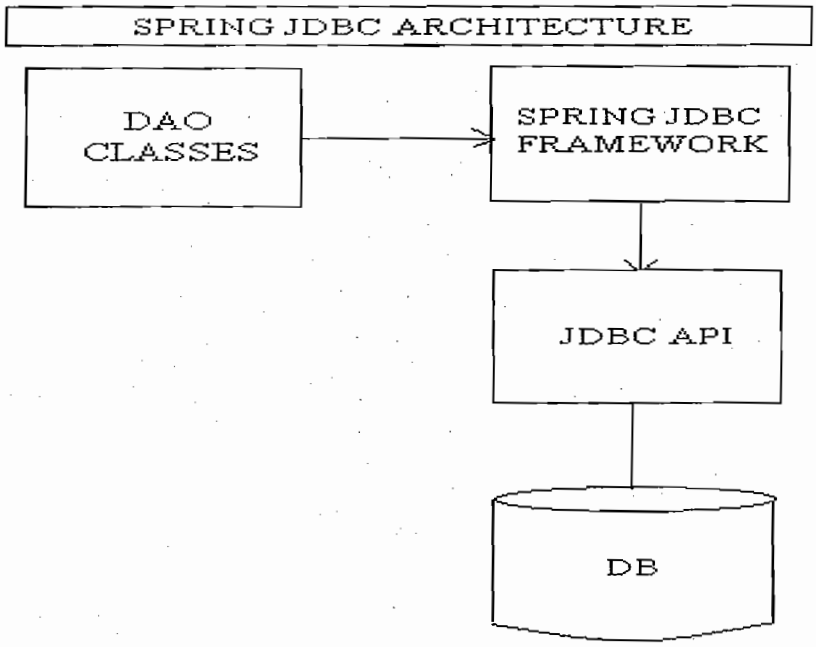
Resource Releasing Logic:

In JDBC

In JDBC API we have to release the DB rescurces (connection.cose();, statement.close(); .....)explicitly.

In Spring:

Spring provided **JdbcTemplate**, which automatically releases the DB resources.



What Spring JDBC does?

Action	Spring	You
Define connection parameters.		X
Open the connection.	X	
Specify the SQL statement.		X
Declare parameters and provide parameter values		X
Prepare and execute the statement.	X	
Set up the loop to iterate through the results (if any).	X	
Do the work for each iteration.		X
Process any exception.	X	

Action	Spring	You
Handle transactions.	X	
Close the connection, statement and resultset.	X	

There are two approaches in building the persistent tier using **SPRING JDBC FRAME/WORK.**

1. **JdbcTemplate .**
2. **Fine Grained ObjectOrientedApproach.**

Spring FrameWork having the capability of establishing the connection ,interacting with DataBase.
Spring FrameWork simplifies the development of JDBC , HIBERNATE , JPA, JDO.
Spring people as provided multiple Templates to interact with DB. Some of the Templates are...

JdbcTemplate
HibernateTemplate
JdoTemplate
JpaTemplate...etc.

All the above are pre-defined Template classes, given by Spring people.
If you want to write a Jdbc code we use JdbcTemplate.

Connection Pooling

Before connection pooling

- ⇨ Opening the connection and closing the connection as and when we required the connection in the application.
- ⇨ This type of connection object utilization is very costly. Because it consumes more time to get connected with the database as and when we require the connection.
- ⇨ This could affect performance of the system and it causes to charge more by the database vendors.
- ⇨ Even multithreaded environment is not suggested, we find the problems scalability, consistency... etc.
- ⇨ Thus we should not hold connection per thread or per application.
- ⇨ To handle these situation servers provided efficient connections management called as connection pooling.
- ⇨ Even some third parties provide connection pooling mechanisms (c3p0) but server provided connection pooling is mostly used in the industries.

Q.) What is connection pooling?

- ⇨ Connection pooling is a mechanism of pre-creating a group of database connections and keeping them in cache memory for use and reuse.
- ⇨ Connection pooling provides the following benefits.
  - o High performance
  - o Efficient resource management.
  - o Cost effectiveness in using database services.

Q.) What is DataSource ?

- ⇨ DataSource is an interface in javax.sql package.
- ⇨ Object oriented representation of Connection pool is nothing but DataSource object.
- ⇨ DataSource is a connection factory for java application.
- ⇨ DataSource is an alternative for DriverManager in providing Database connections to the java application.

Q) Who will provide DataSource implementation class i.e. Who provides Connection pooling mechanism?

- ⇨ Some third party API's implements DataSource. But server provided (means like Tomcat, weblogic, websphere, JBOSS,...etc) Connection pooling mechanism are used in the industries.

Q.) What is pooled Connection? How to acquire pooled Connection in java application?

- ⇨ A database connection acquired by the java application through javax.sql.DataSource is nothing but a pooled connection.
  - o Get DataSource object reference
  - o Call getConnection() method on DataSource object.
  - o Returning the connection to the pool after usage.

Getting DataSource object reference:

- ⇨ DataSource object is registered with Naming service with some name. java program should connect to the naming service using naming API and perform lookup() to get DataSource object reference.

For example,

```
InitialContext ic = new InitialContext();
DataSource ds= ic.lookup("registeredName");
```

Getting pooled Connection:

- ⇨ By calling zero argument getConnection() method on the DataSource object, java application get a pooled connection.

For example,

```
Connection con = ds.getConnection();
```

Returning the connection to the pool after usage:

- ⇨ By calling close() method on connection object it won't close the connection physically with the database. Just it returns the connection back to the pool.

For example,

```
con.close();
```

Q.) How to implement Connection pooling in Tomcat server?

Step 1: copy the jar file of the JDBC driver into Tomcat6.0\lib or Tomcat6.0\common\lib directory. For Example copy classes12.jar file for oracle connectivity.

Step 2: In Tomcat\conf\context.xml file make the following entries.

```
<resource       name="jdbc/myDataSource"
                auth="Container"
                type="javax.sql.DataSource"
                driverclassname="oracle.jdbc.driver.OracleDriver"
                url="jdbc:oracle:thin:@ 115.117.219.10:1521:XE"
                username="system"
                password="tiger" />
```

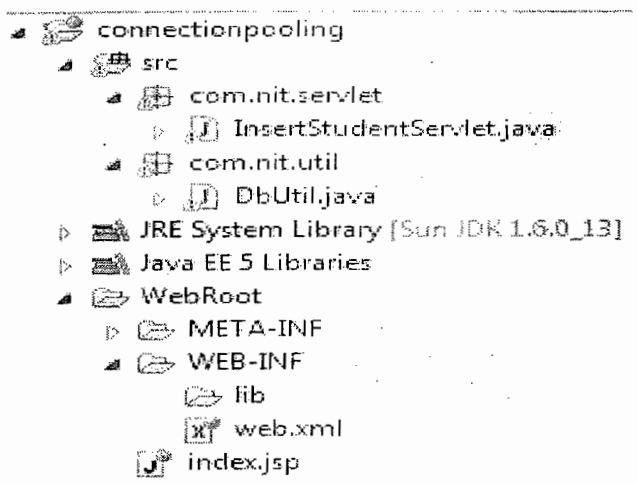
Q.) Example on connection pooling?

Place the following tag in side Tomcat\conf\context.xml file.

```
<resource       name="jdbc/myDataSource"
                auth="Container"
                type="javax.sql.DataSource"
                driverclassname="oracle.jdbc.driver.OracleDriver"
                url="jdbc:oracle:thin:@ 115.117.219.10:1521:XE"
                username="system"
                password="tiger" />
```

create the STUDENT table in the databse:

Name	Course	Age



index.html

```
1. <html>
2.   <body bgcolor="pink" >
3.     <center >
4.     <h1>Student Registration screen</h1>
5.          <form action="InsertStudentServlet">
6.               <br>Name : <input type="text" name="name" >
7.               <br>Course : <input type="text" name="course" >
8.               <br>Age : <input type="text" name="age" >
9.               <br/> <input type="submit" value="Register" >
10.         </form>
11.       </center>
12.     </body>
13.   </html>
```

web.xml

```
1. <web-app >
2.   <servlet>
3.     <servlet-name>InsertStudentServlet</servlet-name>
4.     <servlet-class>com.nit.servlet.InsertStudentServlet</servlet-class>
5.   </servlet>
6.
7.   <servlet-mapping>
8.     <servlet-name>InsertStudentServlet</servlet-name>
9.     <url-pattern>/InsertStudentServlet</url-pattern>
10.    </servlet-mapping>
11.    <welcome-file-list>
12.      <welcome-file>index.jsp</welcome-file>
13.    </welcome-file-list>
14.  </web-app>
```

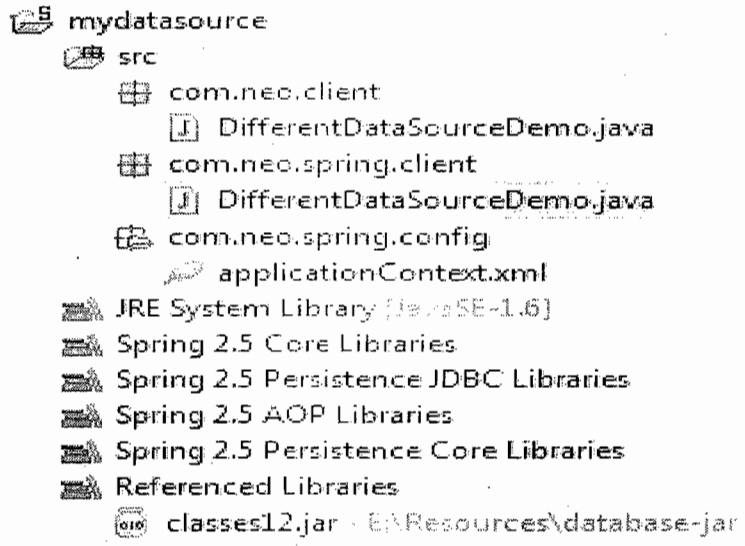
InsertStudentServlet.java

```
1. package com.nit.servlet;
2.
3. import java.io.IOException;
4. import java.io.PrintWriter;
5. import java.sql.Connection;
6. import java.sql.Statement;
7.
8. import javax.servlet.ServletException;
9. import javax.servlet.http.HttpServlet;
10.   import javax.servlet.http.HttpServletRequest;
11.   import javax.servlet.http.HttpServletResponse;
12.
13.   import com.nit.util.DbUtil;
14.
15.   public class InsertStudentServlet extends HttpServlet {
16.
17.         public void doGet(HttpServletRequest request,
18.   HttpServletResponse response) throws ServletException, IOException {
19.
20.               response.setContentType("text/html");
21.               PrintWriter out = response.getWriter();
22.               String name = request.getParameter("name");
23.               String course = request.getParameter("course");
24.               String age = request.getParameter("age");
25.
26.               try {
27.                     Connection connection = DbUtil.getConnection();
28.                     Statement statement = connection.createStatement();
29.                     System.out.println(connection);
30.                     System.out.println(statement);
31.
32.                     statement.executeQuery("INSERT INTO STUDENT VALUES
33.                        ('" + name+ "','" + course + "','" + age + "' )");
34.
35.                     DbUtil.closeConnection(connection);
36.                     out.println("successfully record is inserted");
37.
38.               } catch (Exception e) {
39.                     e.printStackTrace();
40.               out.println("Unable to insert record into the database");
41.               }
42.               out.flush();
43.               out.close();
44.         }
45.
46.   }
```

DbUtil.java

```
1. package com.nit.util;
2.
3. import java.sql.Connection;
4. import java.sql.SQLException;
5.
6. import javax.naming.InitialContext;
```

```
7. import javax.sql.DataSource;
8.
9. public class DbUtil {
10.         private static DataSource dataSource;
11.         static {
12.               try {
13.                     InitialContext initialContext = new InitialContext();
14.                     dataSource = (DataSource)
15.   initialContext.lookup("java:comp/env/jdbc/myDataSource");
16.               } catch (Exception e) {
17.                     e.printStackTrace();
18.               }
19.         }// static
20.
21.         public static Connection getConnection() throws SQLException {
22.               Connection connection = dataSource.getConnection();
23.               return connection;
24.         }// user defined method
25.
26.         public static void closeConnection(Connection connection) {
27.               try {
28.                     if (connection != null) {
29.                           connection.close();
30.                     }
31.               } catch (Exception e) {
32.                     e.printStackTrace();
33.               }
34.         }// user defined method
35.
36.   }
```



com.neo.client.DifferentDataSourceDemo

```
1. package com.neo.client;
2. import java.sql.Connection;
3. import java.sql.ResultSet;
4. import java.sql.Statement;
5.
6. import org.apache.commons.dbcp.BasicDataSource;
7. import org.springframework.jdbc.datasource.DriverManagerDataSource;
8.
9. import com.mchange.v2.c3p0.ComboPooledDataSource;
10.
11.   public class DifferentDataSourceDemo {
12.   public static void main(String[] args) throws Exception {
13.
14.     String query = "SELECT COUNT(*) FROM DUAL";
15.
16.   // Apache people
17.     BasicDataSource apacheDS = new BasicDataSource();
18.     apacheDS.setDriverClassName("oracle.jdbc.driver.OracleDriver");
19.     apacheDS.setUrl("jdbc:oracle:thin:@localhost:1521:XE");
20.     apacheDS.setUsername("system");
21.     apacheDS.setPassword("tiger");
22.     Connection apacheCon = apacheDS.getConnection();
23.     System.out.println("oracle connection getting using apache datasource
24.                                                          : "+ apacheDS);
25.     Statement apacheSt = apacheCon.createStatement();
26.     ResultSet apcheRs = apacheSt.executeQuery(query);
27.     if(apcheRs.next()){
28.         System.out.println("No.Of records in DUAL:"+apcheRs.getInt(1));
29.     }
30.
31.   // spring people
32.     DriverManagerDataSource springDS = new DriverManagerDataSource();
33.     springDS.setDriverClassName("oracle.jdbc.driver.OracleDriver");
34.     springDS.setUrl("jdbc:oracle:thin:@localhost:1521:XE");
35.     springDS.setUsername("system");
```

```
36.     springDS.setPassword("tiger");
37.     Connection springCon = springDS.getConnection();
38.     System.out.println("oracle connection getting using spring datasource
39.                                                          : + springCon);
40.     Statement springSt = springCon.createStatement();
41.     ResultSet springRs = springSt.executeQuery(query);
42.     if(springRs.next()){
43.         System.out.println("No.Of records in DUAL :"+springRs.getInt(1));
44.     }
45.
46.   // c3po people
47.
48.     ComboPooledDataSource c3p0DS = new ComboPooledDataSource();
49.     c3p0DS.setDriverClass("oracle.jdbc.driver.OracleDriver");
50.     c3p0DS.setJdbcUrl("jdbc:oracle:thin:@localhost:1521:XE");
51.     c3p0DS.setUser("system");
52.     c3p0DS.setPassword("tiger");
53.     Connection c3p0Con = c3p0DS.getConnection();
54.     System.outprintln("oracle connection gettting using c3po datasource:"
55.                                                              + c3p0Con);
56.     Statement c3p0St = c3p0Con.createStatement();
57.     ResultSet c3p0Rs = c3p0St.executeQuery(query);
58.     if(c3p0Rs.next()){
59.       System.out.println("No.Ofrecords in DUAL Table:"+c3p0Rs.getInt(1));
60.     }
61.   }
62.   }
```

applicationContext.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3.     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xmlns:p="http://www.springframework.org/schema/p"
5.     xsi:schemaLocation="http://www.springframework.org/schema/beans
6. http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
7.
8.     <!--  Apache datasource configuration -->
9.     <bean id="apacheDatasource"
10.                              class="org.apache.commons.dbcp.BasicDataSource">
11.           <property name="driverClassName"
12.                                  value="oracle.jdbc.driver.OracleDriver"/>
13.       <property name="url" value="jdbc:oracle:thin:@localhost:1521:XE"/>
14.       <property name="username" value="system"/>
15.       <property name="password" value="tiger"/>
16.
17.   </bean>
18.
19.   <!--  Spring datasource configuration -->
20.   <bean id="springDatasource"
21.       class="org.springframework.jdbc.datasource.DriverManagerDataSource"
22.         p:driverClassName="oracle.jdbc.driver.OracleDriver"
23.         p:url="jdbc:oracle:thin:@localhost:1521:XE"
24.         p:username="system" p:password="tiger" />
25.
26.   <!-- c3p0 datasource configuration -->
```

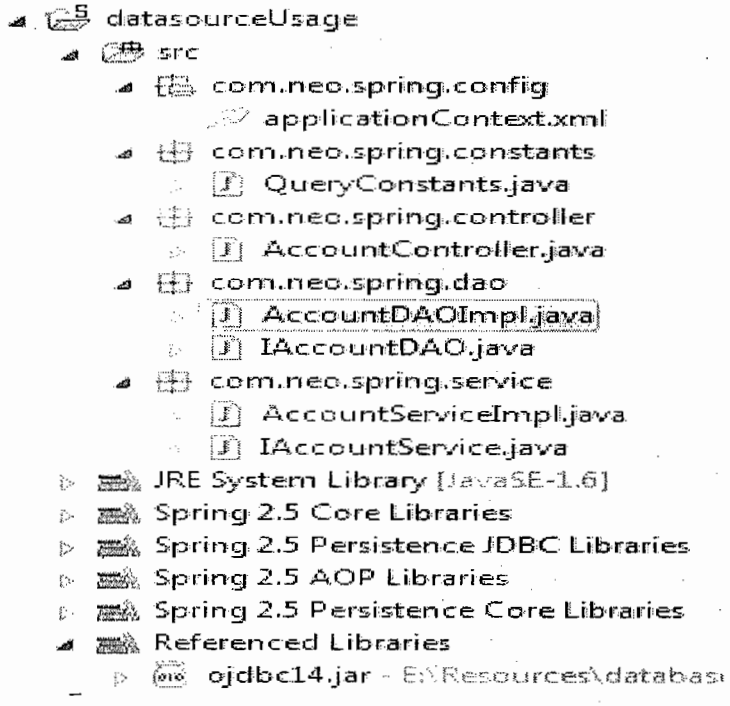
```
27.   <bean id="c3poDatasource"
28.         class="com.mchange.v2.c3p0.ComboPooledDataSource"
29.         p:driverClass="oracle.jdbc.driver.OracleDriver"
30.         p:jdbcUrl="jdbc:oracle:thin:@localhost:1521:XE"
31.         p:user="system" p:password="tiger" />
32.
33.   </beans>
```

NOTE: Here we configured different types of DataSources in applicationContext.xml.
For every DataSource there won't be same properties, they will have their own names.
For e.g., Spring provided ***DriverManagerDataSource*** having property name is 'url' but c3po provided ***ComboPooledDataSource*** having property 'jdbcUrl'. Like this there will be there own property names.

com.neo.spring.client.DifferentDataSourceDemo

```
1. package com.neo.spring.client;
2. import java.sql.Connection;
3. import java.sql.ResultSet;
4. import java.sql.Statement;
5.
6. import javax.sql.DataSource;
7.
8. import org.springframework.context.ApplicationContext;
9. import org.springframework.context.support.ClassPathXmlApplicationContext;
10.
11.   public class DifferentDataSourceDemo {
12.
13.     private static ApplicationContext context =
14.                         new ClassPathXmlApplicationContext(
15.                          "com/neo/spring/config/applicationContext.xml");
16.
17.     public static void main(String[] args) throws Exception {
18.
19.     String query = "SELECT COUNT(*) FROM DUAL";
20.
21.     // Apache people
22.     DataSource apacheDS= (DataSource)context.getBean("apacheDatasource");
23.     Connection apacheCon = apacheDS.getConnection();
24.     System.out.println("oracle connection getting using apache datasource
25.                                                          : "+ apacheDS);
26.     Statement apacheSt = apacheCon.createStatement();
27.     ResultSet apcheRs = apacheSt.executeQuery(query);
28.     if(apcheRs.next()){
29.     System.out.println("No.Of records in DUAL Table:"+apcheRs.getInt(1));
30.     }
31.
32.     // spring people
33.     DataSource springDS= (DataSource)context.getBean("springDatasource");
34.     Connection springCon = springDS.getConnection();
35.     System.out.println("oracle connection getting using spring datasource
36.                                                        :"+ springCon);
37.     Statement springSt = springCon.createStatement();
```

```
38.     ResultSet springRs = springSt.executeQuery(query);
39.     if(springRs.next()){
40.     System.out.println("No.Of records in DUAL Table:+springRs.getInt(1));
41.     }
42.
43.     // c3po people
44.     DataSource c3p0DS = (DataSource)context.getBean("c3poDatasource");
45.     Connection c3p0Con = c3p0DS.getConnection();
46.     System.out.println("oracle connection gettting using c3po datasource
47.                                                          : "+ c3p0Con);
48.     Statement c3p0St = c3p0Con.createStatement();
49.     ResultSet c3p0Rs = c3p0St.executeQuery(query);
50.     if(c3p0Rs.next()){
51.     System.out.println("No.Of records in DUAL Table: "+c3p0Rs.getInt(1));
52.     }
53.
54.   }
55.  }
```



applicationContext.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3.     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xmlns:p="http://www.springframework.org/schema/p"
5.     xsi:schemaLocation="http://www.springframework.org/schema/beans
6. http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
7.
8.     <!--  Apache datasource configuration -->
9.     <bean id="apacheDatasourceRef"
10.                              class="org.apache.commons.dbcp.BasicDataSource">
11.           <property name="driverClassName"
12.                                 value="oracle.jdbc.driver.OracleDriver" />
13.      <property name="url" value="jdbc:oracle:thin:@localhost:1521:XE" />
14.      <property name="username" value="system" />
15.      <property name="password" value="tiger" />
16.    </bean>
17.
18.    <bean id="accountDAORef" class="com.neo.spring.dao.AccountDAOImpl">
19.          <property name="dataSource" ref="apacheDatasourceRef"></property>
20.    </bean>
21.
22.    <bean id="accountServiceRef"
23.                          class="com.neo.spring.service.AccountServiceImpl">
24.          <property name="accountDAO" ref="accountDAORef"></property>
25.    </bean>
26.
27.    </beans>
```

QueryConstants.java

```
1. package com.neo.spring.constants;
2.
```

```
3. public interface QueryConstants {
4.     public static final String UPDATE_QUERY =
5.                                 "UPDATE ACCOUNT SET NAME=? WHERE ACCNO=?";
6. }
```

IAccountDAO.java

```
1. package com.neo.spring.dao;
2.
3. public interface IAccountDAO {
4.     public void update(int accno, String name);
5. }
```

AccountDAOImpl.java

```
1. package com.neo.spring.dao;
2.
3. import java.sql.Connection;
4. import java.sql.PreparedStatement;
5. import javax.sql.DataSource;
6. import com.neo.spring.constants.QueryConstants;
7.
8. public class AccountDAOImpl implements IAccountDAO {
9.     @Override
10.         public void update(int accno, String name) {
11.             Connection connection = null;
12.             PreparedStatement statement = null;
13.             try {
14.                 connection = dataSource.getConnection();
15.                 statement = connection
16.                         .prepareStatement(QueryConstants.UPDATE_QUERY);
17.                 statement.setString(1, name);
18.                 statement.setInt(2, accno);
19.                 statement.executeUpdate();
20.
21.             } catch (Exception e) {
22.                 e.printStackTrace();
23.             } finally {
24.                 try {
25.                     connection.close();
26.                     statement.close();
27.                 } catch (Exception e2) {
28.                     e2.printStackTrace();
29.                 }
30.
31.             }
32.
33.         }
34.
35.         private DataSource dataSource;
36.
37.         public void setDataSource(DataSource dataSource) {
38.             this.dataSource = dataSource;
39.         }
40.     }
```

IAccountService.java

```
1. package com.neo.spring.service;
2.
3. public interface IAccountService {
4.     public void modify(int accno, String name);
5. }
```

AccountServiceImpl.java

```
1. package com.neo.spring.service;
2. import com.neo.spring.dao.IAccountDAO;
3. public class AccountServiceImpl implements IAccountService {
4.     @Override
5.     public void modify(int accno, String name) {
6.           accountDAO.update(accno, name);
7.     }
8.
9.     private IAccountDAO accountDAO;
10.
11.          public void setAccountDAO(IAccountDAO accountDAO) {
12.                this.accountDAO = accountDAO;
13.          }
14.    }
```

AccountController.java

```
1. package com.neo.spring.controller;
2.
3. import org.springframework.context.ApplicationContext;
4. import org.springframework.context.support.ClassPathXmlApplicationContext;
5. import com.neo.spring.service.IAccountService;
6. public class AccountController {
7.    private static ApplicationContext context =
8.                        new ClassPathXmlApplicationContext(
9.                "com/neo/spring/config/applicationContext.xml");
10.
11.          public static void main(String[] args) {
12.                IAccountService service= (IAccountService)
13.                                        context.getBean("accountServiceRef");
14.                service.modify(1001,"somasekhar");
15.                System.out.println("Account is updated successfully");
16.          }
17.    }
```

Output:

```
Account is updated successfully
```

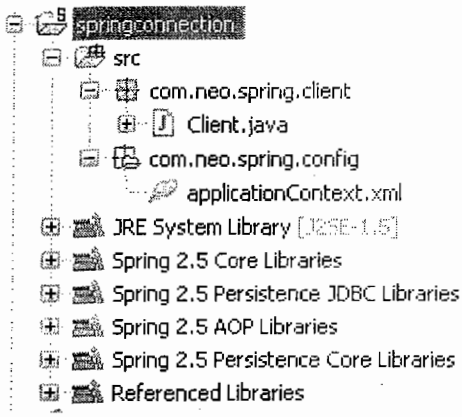
Initially Account Table:

ACCNO	NAME	BALANCE
1001	sekhar	2500

After Updating record:

ACCNO	NAME	BALANCE
1001	somasekhar	2500

NOTE : If we observe above application we only handling the exception, allocating the resources and releasing the resources. But these all are done by using jdbcTemplate.



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"
      xmlns:p="http://www.springframework.org/schema/p"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

      <!-- Class.forName("oracle.jdbc.driver.OracleDriver") -->
      <bean class="oracle.jdbc.driver.OracleDriver"></bean>

      <!-- Connection connection= DriverManager.getConnection(
"jdbc:oracle:thin:@localhost:1521:server","scott","tiger") -->
      <bean id="conRef" class="java.sql.DriverManager"
                              factory- method="getConnection">
            <constructor-arg
              value="jdbc:oracle:thin:@localhost:1521:server"></constructor-arg>
            <constructor-arg value="scott"></constructor-arg>
            <constructor-arg value="tiger"></constructor-arg>
      </bean>

      <!-- Statement statement=connection.createStatement() -->
      <bean id="stRef" class="java.sql.Statement" factory-bean="conRef"
                              factory-method="createStatement"></bean>

      <!-- ResultSet resultSet =
      statement.executeQuery("SELECT BALANCE FROM ACCOUNT WHERE ACCNO=1001") -->
      <bean id="rsRef" class="java.sql.ResultSet" factory-bean="stRef"
                                          factory-method="executeQuery">
            <constructor-arg
                  value="SELECT BALANCE FROM ACCOUNT WHERE ACCNO=1001" />
      </bean>
</beans>
```

Client.java

```
package com.neo.spring.client;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.Statement;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
public class Client {
	private static ApplicationContext context=new
ClassPathXmlApplicationContext("com/neo/spring/config/applicationContext.xml");
	public static void main(String[] args) throws Exception{
		Connection connection=(Connection)context.getBean("conRef");
		System.out.println(connection);

		Statement statement=(Statement)context.getBean("stRef");
		System.out.println(statement);

		ResultSet resultSet=(ResultSet)context.getBean("rsRef");
		if(resultSet.next()){
			double balance=resultSet.getDouble(1);
			System.out.println("balance:"+balance);
		}
	}
}
```

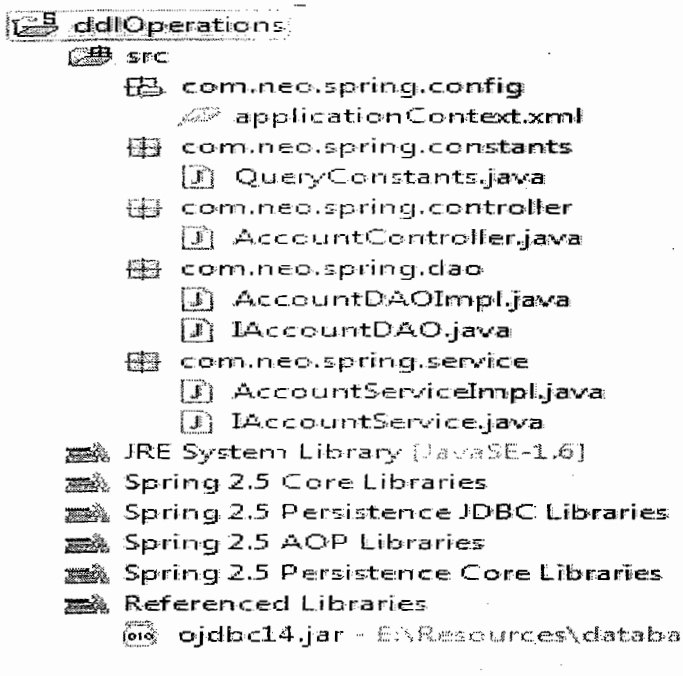
Account.Table

ACCNO	NAME	BALANCE
1001	sekhar	2500

We have this record in the account table.

Output

```
oracle.jdbc.driver.OracleConnection@992bae
oracle.jdbc.driver.OracleStatement@6f9b8e
balance:2500.0
```



applicationContext.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3.     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xmlns:p="http://www.springframework.org/schema/p"
5.     xsi:schemaLocation="http://www.springframework.org/schema/beans
6. http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
7.
8. <!--  Apache datasource configuration -->
9.     <bean id="apacheDatasourceRef"
10.                          class="org.apache.commons.dbcp.BasicDataSource">
11.      <property name="driverClassName"
12.                                  value="oracle.jdbc.driver.OracleDriver" />
13.      <property name="url" value="jdbc:oracle:thin:@localhost:1521:XE" />
14.      <property name="username" value="system" />
15.      <property name="password" value="tiger" />
16.    </bean>
17.
18.    <bean id="jdbcTemplateRef"
19.                        class="org.springframework.jdbc.core.JdbcTemplate" >
20.          <property name="dataSource" ref="apacheDatasourceRef" />
21.    </bean>
22.
23.    <bean id="accountDAORef" class="com.neo.spring.dao.AccountDAOImpl">
24.          <property name="dataSource" ref=" jdbcTemplateRef"></property>
25.    </bean>
26.
27.    <bean id="accountServiceRef"
28.                          class="com.neo.spring.service.AccountServiceImpl">
29.          <property name="accountDAO" ref="accountDAORef"></property>
30.    </bean>
31.
32.    </beans>
```

QueryConstants.java

```
1. package com.neo.spring.constants;
2. public interface QueryConstants {
3.    public static String CREATE_QUERY = "CREATE TABLE ACCOUNT (ACCNO
4.                          NUMBER(5), NAME VARCHAR2(10), BAL NUMBER(8,2))";
5.    public static String ALTER_QUERY =
6.                                    "ALTER TABLE ACCOUNT DROP COLUMN BAL";
7.    public static String TRUNCATE_QUERY = "TRUNCATE TABLE ACCOUNT";
8.    public static String DROP_QUERY = "DROP TABLE ACCOUNT ";
9. public static String DRL_QUERY = "SELECT COUNT(*) FROM DUAL";
10.   public static String DML_QUERY = "UPDATE ACCOUNT SET NAME='sekhar'";
11.
12.   }
```

IAccountDAO.java

```
1. package com.neo.spring.dao;
2. public interface IAccountDAO {
3.      public void create();
4.      public void alter();
5.      public void truncate();
6.      public void drop();
7.      public void dml();
8.      public void drl();
9.
10.   }
```

AccountDAOImpl.java

```
1. package com.neo.spring.dao;
2. import org.springframework.jdbc.core.JdbcTemplate;
3. import com.neo.spring.constants.QueryConstants;
4. public class AccountDAOImpl implements IAccountDAO {
5.          @Override
6.          public void create() {
7.          template.execute(QueryConstants.CREATE_QUERY);
8.    }
9.
10.         @Override
11.         public void alter() {
12.               template.execute(QueryConstants.ALTER_QUERY);
13.         }
14.
15.         @Override
16.         public void truncate() {
17.               template.execute(QueryConstants.TRUNCATE_QUERY);
18.         }
19.
20.         @Override
21.         public void drop() {
22.               template.execute(QueryConstants.DROP_QUERY);
23.         }
24.
25.         @Override
26.         public void dml() {
27.               template.execute(QueryConstants.DML_QUERY);
28.         }
```

```
29.
30.         @Override
31.         public void drl() {
32.               template.execute(QueryConstants.DRL_QUERY);
33.         }
34.
35.         private JdbcTemplate template;
36.
37.         public void setTemplate(JdbcTemplate template) {
38.               this.template = template;
39.         }
40.   }
```

IAccountService.java

```
1. package com.neo.spring.service;
2. public interface IAccountService {
3.       public void create();
4.       public void alter();
5.       public void truncate();
6.       public void drop();
7.       public void dml();
8.       public void drl();
9. }
```

AccountServiceImpl.java

```
1. package com.neo.spring.service;
2. import com.neo.spring.dao.IAccountDAO;
3. public class AccountServiceImpl implements IAccountService {
4.
5.          @Override
6.          public void create() {
7.                accountDAO.create();
8.    }
9.
10.         @Override
11.         public void alter() {
12.               accountDAO.alter();
13.         }
14.
15.         @Override
16.         public void truncate() {
17.               accountDAO.truncate();
18.         }
19.
20.         @Override
21.         public void drop() {
22.               accountDAO.drop();
23.         }
24.
25.         @Override
26.         public void dml() {
27.               accountDAO.dml();
28.         }
29.
30.         @Override
```

```
31.         public void drl() {
32.               accountDAO.drl();
33.         }
34.
35.         private IAccountDAO accountDAO;
36.
37.         public void setAccountDAO(IAccountDAO accountDAO) {
38.               this.accountDAO = accountDAO;
39.         }
40.   }
```

AccountController.java

```
1. package com.neo.spring.controller;
2. import org.springframework.context.ApplicationContext;
3. import org.springframework.context.support.ClassPathXmlApplicationContext;
4. import com.neo.spring.service.IAccountService;
5.
6. public class AccountController {
7.    private static ApplicationContext context = new
8.                ClassPathXmlApplicationContext(
9.                "com/neo/spring/config/applicationContext.xml");
10.
11.         public static void main(String[] args) {
12.               IAccountService service= (IAccountService)
13.                                    context.getBean("accountServiceRef");
14.   //          service.create();
15.   //          service.alter();
16.   //          service.truncate();
17.   //          service.drop();
18.   //          service.drl();
19.   //          service.dml();
20.
21.               System.out.println("success");
22.          }
23.   }
```

Ouput:

After `dao.create();` is executed then Account table is created.

ACCNO	NAME	BAL

Now go to database, insert one record into Account table.

ACCNO	NAME	BAL
1001	sekhar	2500

After execution of `dao.truncate();`

ACCNO	NAME	BAL

After execution of `dao.alter();`

ACCNO	NAME

```
After execution of dao.drop(); Account table will be dropped.
SQL> SELECT * FROM ACCOUNT;
ERROR at line 1:
ORA-00942: table or view does not exist
```

NOTE : If we observe dml(), drl() method of AccountDaoIml class, those are not returning any results. Because **JdbcTemplate.execute(String query)** method return type is void. So, we should not use **JdbcTemplate.execute(String query)** method to perform DML, DRL operations.

JdbcTemplate.execute(String query) method only meant for performing DDL operations. But generally we don't use this method in projects. Because first Database design(Data model) developed for the project.

Q) What is the need of Callback beans (ConnectionCallback, StatementCallback, PreparedStatementCallback, CallableStatementCallback) ?

- When **JdbcTemplate** is not providing the features of jdbc technology Resources (**Connection**, **Statement**, **PreparedStatement**, **CallableStatement** etc) we will go for Callback mechanism.
- In this mechanism we will directly get Jdbc technology resources, so that we can use all the features of Jdbc technology.
- In this approach whatever the resources we created (from spring given resources) we have to close them physically.
- In this callback mechanism we have to write the code to execute the queries.

Q) What is the need of creators (PreparedStatementCreator, CallableStatementCreator etc)?

Ans: It is similar to callback mechanism except the following differences:

- Here we just create the resources (**PreparedStatement**, **CallableStatement** ...etc) and return to spring container but we don't close them.
- Here we no need to write the code to execute the queries.

public Object execute(ConnectionCallback):-

ConnectionCallback is an interface. It has **doInConnection(Connection con)**, it returns Object.
If we want to access this method then for a simple java class we have to implement **ConnectionCallback** interface and override **doInConnection(Connection)** method.
It allows to execute any number of operations on a single Connection, using any type and number of Statements.It does not need to care about activating or closing the Connection, or handling transactions.

public Object execute(StatementCallback):-

StatementCallback is an interface. It has **doInStatement(Statement stmt),** it returns Object.
If we want to access this method then for a simple java class we have to implement StatementCallback interface and override **doInStatement(Statement)** method. Allows to execute any number of operations on a single Statement.

It does not need to care about closing the Statement or the Connection, or about handling transactions: this will all be handled by Spring's **JdbcTemplate**.

If any **ResultSets** are opened then we should close them. Because spring will automatically close the resource whatever it opens the resources.

public Object execute(String sql, PreparedStatementCallback):-

PreparedStatementCallback is an interface. It has **doInPreparedStatement(PreparedStatement stmt),** it returns Object. If we want to access this method then for a simple java class we have to implement **PreparedStatementCallback** interface and override **doInStatement(Statement)** method. Allows to execute any number of operations on a single **PreparedStatement**.

public Object execute(String callString, CallableStatementCallback):-

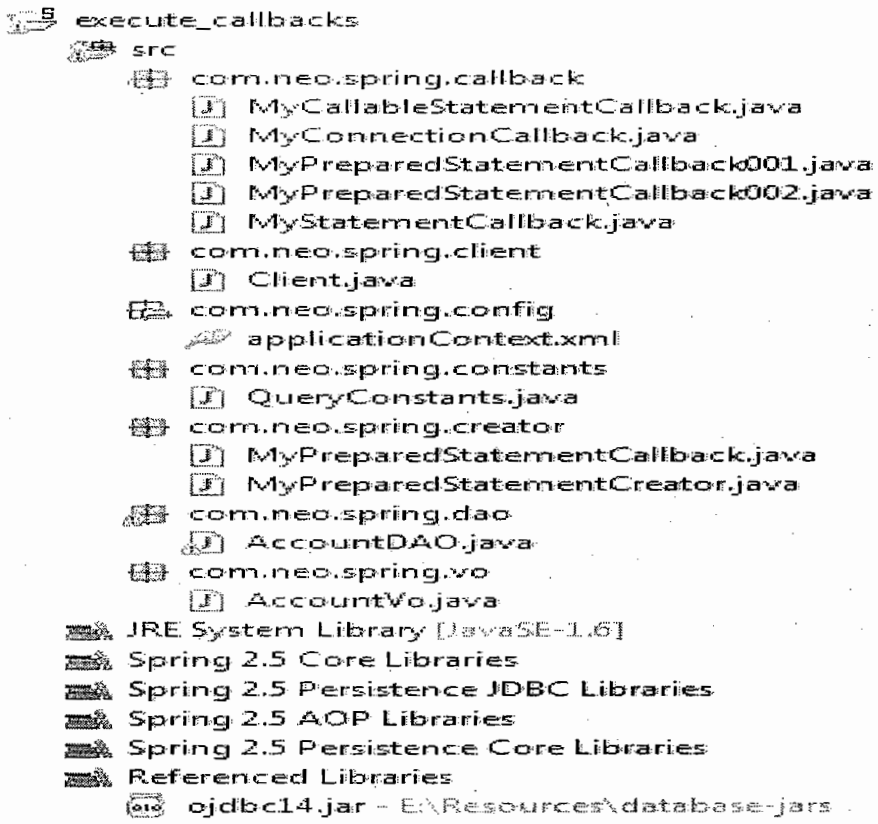
CallableStatementCallback is an interface. It has **doInCallableStatement(CallableStatement cs)**. If we want to access this method then we have to implement **CallableStatementCallback** interface to our class and override **doInCallableStatement**() method. It allows to execute any number of operations on a single **CallableStatement**. We need not to close any connection or statements. Spring will close the Statement object after the callback returned, but this does not necessarily imply that the **ResultSet** resources will be closed: the Statement objects might get pooled by the connection pool, with close calls only returning the object to the pool but not physically closing the resources.

public Object execute(PreparedStatementCreator, PreparedStatementCallback):-

This interface creates a **PreparedStatement** given a connection, provided by the **JdbcTemplate** class. Implementations are responsible for providing SQL and any necessary parameters.

Implementations *do not* need to concern themselves with **SQLExceptions** that may be thrown from operations they attempt. The **JdbcTemplate** class will catch and handle **SQLExceptions** appropriately.

A **PreparedStatementCreator** should also implement the SqlProvider interface if it is able to provide the SQL it uses for **PreparedStatement** creation. This allows for better contextual information in case of exceptions.



applicationContext.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3.     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xmlns:p="http://www.springframework.org/schema/p"
5.     xsi:schemaLocation="http://www.springframework.org/schema/beans
6. http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
7.
8. <!--  Apache datasource configuration -->
9.     <bean id="apacheDatasourceRef"
10.                         class="org.apache.commons.dbcp.BasicDataSource">
11.      <property name="driverClassName"
12.                                 value="oracle.jdbc.driver.OracleDriver" />
13.      <property name="url" value="jdbc:oracle:thin:@localhost:1521:XE" />
14.      <property name="username" value="system" />
15.      <property name="password" value="tiger" />
16.    </bean>
17.
```

```
18.   <bean id="jdbcTemplateRef"
19.                       class="org.springframework.jdbc.core.JdbcTemplate" >
20.         <property name="dataSource" ref="apacheDatasourceRef" />
21.   </bean>
22.
23.   <bean id="accountDAORef" class="com.neo.spring.dao.AccountDAO">
24.         <property name="dataSource" ref=" jdbcTemplateRef"></property>
25.   </bean>
26.
27.   </beans>
```

QueryConstants.java

```
1. package com.neo.spring.constants;
2.
3. public interface QueryConstants {
4.    public static final String CONNECTION_CALLBACK = "UPDATE ACCOUNT SET
5.                                        NAME='somasekhar' WHERE ACCNO=1001";
6.    public static final String STATEMENT_CALLBACK = "SELECT * FROM ACCOUNT
7.                                                      WHERE ACCNO=1001";
8.    public static final String PS_CALLBACK = "INSERT INTO ACCOUNT
9.                                                        VALUES(?,?,?)";
10.   public static final String PS_CREATOR = "SELECT * FROM ACCOUNT
11.                                                         WHERE BAL>?";
12.   }
```

MyCallableStatementCallback.java

```
1. package com.neo.spring.callback;
2.
3. import java.sql.CallableStatement;
4. import java.sql.SQLException;
5. import java.sql.Types;
6.
7. import org.springframework.dao.DataAccessException;
8. import org.springframework.jdbc.core.CallableStatementCallback;
9.
10.   import com.neo.spring.vo.AccountVo;
11.
12.   public class MyCallableStatementCallback implements
13.                                       CallableStatementCallback {
14.         private AccountVo accountVo;
15.         public void setAccountVo(AccountVo accountVo) {
16.               this.accountVo = accountVo;
17.         }
18.   @Override
19.   public Object doInCallableStatement(CallableStatement cs)
20.                            throws SQLException, DataAccessException {
21.               String result = null;
22.               try {
23.                     cs.setInt(1, accountVo.getAccno());
24.                     cs.setString(2, accountVo.getName());
25.                     cs.setDouble(3, accountVo.getBalance());
26.                     cs.registerOutParameter(4, Types.VARCHAR);
27.                     cs.execute();
28.                     result =cs.getString(4);
29.               } catch (SQLException e) {
```

```
30.                     e.printStackTrace();
31.                 }
32.
33.         return result;
34.     }
35.     }
```

MyConnectionCallback.java

```
1. package com.neo.spring.callback;
2.
3. import java.sql.Connection;
4. import java.sql.SQLException;
5. import java.sql.Statement;
6.
7. import org.springframework.dao.DataAccessException;
8. import org.springframework.jdbc.core.ConnectionCallback;
9.
10.   import com.neo.spring.constants.QueryConstants;
11.
12.   public class MyConnectionCallback implements ConnectionCallback {
13.
14.   @Override
15.   public Object doInConnection(Connection con) throws SQLException,
16.                                                       DataAccessException {
17.         int ra = 0;
18.         Statement statement = null;
19.         try {
20.           statement = con.createStatement();
21.           ra=statement.executeUpdate(QueryConstants.CONNECTION_CALLBACK);
22.         } catch (SQLException e) {
23.               e.printStackTrace();
24.         } finally {
25.               statement.close();
26.         }
27.
28.         return ra;
29.   }
30.   }
```

MyPreparedStatementCallback001.java

```
1. package com.neo.spring.callback;
2.
3. import java.sql.PreparedStatement;
4. import java.sql.SQLException;
5.
6. import org.springframework.dao.DataAccessException;
7. import org.springframework.jdbc.core.PreparedStatementCallback;
8.
9. public class MyPreparedStatementCallback001 implements
10.                                     PreparedStatementCallback {
11.
12.         @Override
13.         public Object doInPreparedStatement(PreparedStatement ps)
14.                     throws SQLException, DataAccessException {
```

```
15.               int ra=0;
16.               try {
17.                     ps.setInt(1, 1002);
18.                     ps.setString(2, "yerragudi");
19.                     ps.setDouble(3, 6800.0);
20.                     ra = ps.executeUpdate();
21.               } catch (SQLException e) {
22.                     e.printStackTrace();
23.               }
24.
25.
26.               return ra;
27.         }
28.
29.   }
```

MyPreparedStatementCallback002.java

```
1. package com.neo.spring.callback;
2.
3. import java.sql.PreparedStatement;
4. import java.sql.SQLException;
5.
6. import org.springframework.dao.DataAccessException;
7. import org.springframework.jdbc.core.PreparedStatementCallback;
8.
9. import com.neo.spring.vo.AccountVo;
10.
11.   public class MyPreparedStatementCallback002 implements
12.               PreparedStatementCallback {
13.         private AccountVo accountVo;
14.
15.         public void setAccountVo(AccountVo accountVo) {
16.               this.accountVo = accountVo;
17.         }
18.
19.         @Override
20.         public Object doInPreparedStatement(PreparedStatement ps)
21.                     throws SQLException, DataAccessException {
22.               int ra = 0;
23.               try {
24.                     ps.setInt(1, accountVo.getAccno());
25.                     ps.setString(2, accountVo.getName());
26.                     ps.setDouble(3, accountVo.getBalance());
27.                     ra = ps.executeUpdate();
28.               } catch (SQLException e) {
29.                     e.printStackTrace();
30.               }
31.
32.               return ra;
33.         }
34.
35.   }
```

MyStatementCallback.java

```
1. package com.neo.spring.callback;
```

```
2.
3. import java.sql.ResultSet;
4. import java.sql.SQLException;
5. import java.sql.Statement;
6.
7. import org.springframework.dao.DataAccessException;
8. import org.springframework.jdbc.core.StatementCallback;
9.
10.   import com.neo.spring.constants.QueryConstants;
11.   import com.neo.spring.vo.AccountVo;
12.
13.   public class MyStatementCallback implements StatementCallback {
14.         @Override
15.         public Object doInStatement(Statement statement) throws
16.                               SQLException, DataAccessException {
17.               AccountVo accountVo = new AccountVo();
18.               ResultSet rs = null;
19.               try {
20.
21.              rs=statement.executeQuery(QueryConstants.STATEMENT_CALLBACK);
22.                     if (rs.next()) {
23.                           accountVo.setAccno(rs.getInt(1));
24.                           accountVo.setName(rs.getString(2));
25.                           accountVo.setBalance(rs.getDouble(3));
26.                     }
27.               } catch (SQLException e) {
28.                     e.printStackTrace();
29.               }finally{
30.                     rs.close();
31.               }
32.               return accountVo;
33.         }
34.   }
```

MyPreparedStatementCallback.java

```
1. package com.neo.spring.creator;
2.
3. import java.sql.PreparedStatement;
4. import java.sql.ResultSet;
5. import java.sql.SQLException;
6. import java.util.ArrayList;
7.
8. import org.springframework.dao.DataAccessException;
9. import org.springframework.jdbc.core.PreparedStatementCallback;
10.
11.   import com.neo.spring.vo.AccountVo;
12.
13.   public class MyPreparedStatementCallback implements
14.                                         PreparedStatementCallback {
15.   @Override
16.   public Object doInPreparedStatement(PreparedStatement ps)
17.               throws SQLException, DataAccessException {
18.         ArrayList<AccountVo> list = new ArrayList<AccountVo>();
19.         ResultSet rs = null;
20.         try {
```

```
21.               ps.setDouble(1, 1050.0);
22.               rs = ps.executeQuery();
23.               while(rs.next()){
24.                     AccountVo accountVo = new AccountVo();
25.                     accountVo.setAccno(rs.getInt(1));
26.                     accountVo.setName(rs.getString(2));
27.                     accountVo.setBalance(rs.getDouble(3));
28.                     list.add(accountVo);
29.
30.               }
31.         } catch (SQLException e) {
32.               e.printStackTrace();
33.         }finally{
34.               rs.close();
35.         }
36.         return list;
37.   }
38.   }
```

MyPreparedStatementCreator.java

```
1. package com.neo.spring.creator;
2.
3. import java.sql.Connection;
4. import java.sql.PreparedStatement;
5. import java.sql.SQLException;
6.
7. import org.springframework.jdbc.core.PreparedStatementCreator;
8.
9. import com.neo.spring.constants.QueryConstants;
10.
11.   public class MyPreparedStatementCreator implements
12.                                             PreparedStatementCreator{
13.   @Override
14.   public PreparedStatement createPreparedStatement(Connection connection)
15.                     throws SQLException {
16.         return connection.prepareStatement(QueryConstants.PS_CREATOR);
17.   }
18.   }
```

AccountVo.java

```
1. package com.neo.spring.vo;
2.
3. public class AccountVo {
4.    private int accno;
5.    private String name;
6.    private double balance;
7.
8.    public int getAccno() {
9.          return accno;
10.         }
11.
12.         public void setAccno(int accno) {
13.               this.accno = accno;
14.         }
15.
```

```
16.         public String getName() {
17.               return name;
18.         }
19.
20.         public void setName(String name) {
21.               this.name = name;
22.         }
23.
24.         public double getBalance() {
25.               return balance;
26.         }
27.
28.         public void setBalance(double balance) {
29.               this.balance = balance;
30.         }
31.
32.   }
```

AccountDAO.java

```
1. package com.neo.spring.dao;
2.
3. import java.sql.PreparedStatement;
4. import java.sql.SQLException;
5. import java.util.List;
6.
7. import org.springframework.dao.DataAccessException;
8. import org.springframework.jdbc.core.JdbcTemplate;
9. import org.springframework.jdbc.core.PreparedStatementCallback;
10.
11.   import com.neo.spring.callback.MyCallableStatementCallback;
12.   import com.neo.spring.callback.MyConnectionCallback;
13.   import com.neo.spring.callback.MyPreparedStatementCallback001;
14.   import com.neo.spring.callback.MyPreparedStatementCallback002;
15.   import com.neo.spring.callback.MyStatementCallback;
16.   import com.neo.spring.constants.QueryConstants;
17.   import com.neo.spring.creator.MyPreparedStatementCallback;
18.   import com.neo.spring.creator.MyPreparedStatementCreator;
19.   import com.neo.spring.vo.AccountVo;
20.
21.   public class AccountDAO{
22.
23.         private JdbcTemplate template;
24.
25.         public void setTemplate(JdbcTemplate template) {
26.               this.template = template;
27.         }
28.
29.         public int usingConnectionCallback(){
30.          int ra = (Integer) template.execute(new MyConnectionCallback());
31.          return ra;
32.         }
33.
34.         public AccountVo usingStatementCallback(){
35.               AccountVo accountVo = (AccountVo)
36.                                     template.execute(new MyStatementCallback());
```

```
37.             return accountVo;
38.         }
39.
40.         public int usingPreparedStatementCallback001(){
41.           int ra = (Integer) template.execute(QueryConstants.PS_CALLBACK,
42.                                     new MyPreparedStatementCallback001());
43.           return ra;
44.         }
45.
46.        public int usingPreparedStatementCallback002(AccountVo accountVo){
47.
48.         MyPreparedStatementCallback002 mpsc = new
49.                                          MyPreparedStatementCallback002();
50.         mpsc.setAccountVo(accountVo);
51.
52.        int ra=(Integer)template.execute(QueryConstants.PS_CALLBACK,mpsc);
53.          return ra;
54.         }
55.
56.         public int usingPreparedStatementCallback003(final AccountVo
57.                                                              accountVo){
58.
59.         int ra = (Integer) template.execute(QueryConstants.PS_CALLBACK,
60.                   new PreparedStatementCallback() {
61.
62.               @Override
63.               public Object doInPreparedStatement(PreparedStatement ps)
64.                           throws SQLException, DataAccessException {
65.                           int ra=0;
66.                 try {
67.                   ps.setInt(1, accountVo.getAccno());
68.                   ps.setString(2, accountVo.getName());
69.                   ps.setDouble(3, accountVo.getBalance());
70.                   ra = ps.executeUpdate();
71.                 } catch (SQLException e) {
72.                    e.printStackTrace();
73.                 }
74.                   return ra;
75.               }
76.
77.             });
78.
79.               return ra;
80.        }
81.
82.        public String usingCallableStatementCallback(AccountVo
83.                                                          accountVo){
84.
85.               MyCallableStatementCallback mcsc = new
86.                                          MyCallableStatementCallback();
87.               mcsc.setAccountVo(accountVo);
88.
89.               String result = (String)template.execute(
90.                           "{CALL ACCOUNT_INSERT_PROC(?,?,?,?)}", mcsc);
91.               return result;
```

```
92.         }
93.
94.         public List<AccountVo> usingPreparedStatementCreator(){
95.               List<AccountVo> accountVos = (List<AccountVo>)
96.                 template.execute(new MyPreparedStatementCreator(),
97.                                        new MyPreparedStatementCallback());
98.               return accountVos;
99.         }
100.  }
```

Client.java

```
1. package com.neo.spring.client;
2.
3. import java.util.List;
4.
5. import org.springframework.context.ApplicationContext;
6. import org.springframework.context.support.ClassPathXmlApplicationContext;
7.
8. import com.neo.spring.dao.AccountDAO;
9. import com.neo.spring.vo.AccountVo;
10.
11.   public class Client {
12.         private static ApplicationContext context =
13.                   new ClassPathXmlApplicationContext(
14.                     "com/neo/spring/config/applicationContext.xml");
15.
16.         public static void main(String[] args) {
17.               AccountDAO accountDAO = (AccountDAO)
18.                                       context.getBean("accountDAORef");
19.
20.               sop("\n----usingConnectionCallback----");
21.               int ra = accountDAO.usingConnectionCallback();
22.               sop("No.of rows updated : "+ra);
23.
24.               sop("\n----usingStatementCallback----");
25.               sop("Account Details are ....");
26.               AccountVo accountVoA = accountDAO.usingStatementCallback();
27.               sop(accountVoA.getAccno());
28.               sop(accountVoA.getName());
29.               sop(accountVoA.getBalance());
30.
31.               sop("\n----usingPreparedStatementCallback----");
32.               ra = accountDAO.usingPreparedStatementCallback001();
33.               sop("No.of rows inserted : "+ra);
34.
35.               sop("\n----usingPreparedStatementCallback----");
36.               AccountVo accountVoB = new AccountVo();
37.               accountVoB.setAccno(1003);
38.               accountVoB.setName("l.n.rao");
39.               accountVoB.setBalance(5600.0);
40.              ra=accountDAO.usingPreparedStatementCallback002(accountVoB);
41.               sop("No.of rows inserted : "+ra);
42.
43.               sop("\n----usingPreparedStatementCallback003----");
44.               AccountVo accountVoC = new AccountVo();
```

```
45.                 accountVoC.setAccno(1004);
46.                 accountVoC.setName("kesavareddy");
47.                 accountVoC.setBalance(5600.0);
48.                ra=accountDAO.usingPreparedStatementCallback003(accountVoC);
49.                 sop("No.of rows inserted : "+ra);
50.
51.                 sop("----usingCallableStatementCallback----");
52.                 AccountVo accountVoD = new AccountVo();
53.                 accountVoD.setAccno(1005);
54.                 accountVoD.setName("yerragudi");
55.                 accountVoD.setBalance(4600.0);
56.                 String result =
57.                      accountDAO.usingCallableStatementCallback(accountVoD);
58.                 sop("Procedure execution status :"+result);
59.
60.                 sop("----usingPreparedStatementCreator-----");
61.                 List<AccountVo> accountVos =
62.                                  accountDAO.usingPreparedStatementCreator();
63.                 for(AccountVo accountVo : accountVos){
64.                       sop(accountVo.getAccno());
65.                       sop(accountVo.getName());
66.                       sop(accountVo.getBalance());
67.
68.                 }
69.           }
70.           public static void sop(Object object){
71.                 System.out.println(object);
72.           }
73.     }
```

PROCEDURE:

```
1. CREATE OR REPLACE PROCEDURE SYSTEM.ACCOUNT_INSERT_PROC (p_accno number, p_name
   varchar2, p_bal number, p_result out varchar2) IS
2.
3. BEGIN
4.
5.     insert into account values(p_accno, p_name, p_bal);
6.     commit;
7.     p_result:='success';
8.
9.   EXCEPTION                    -- exception handlers begin
10.   --WHEN ZERO_DIVIDE THEN  -- handles 'division by zero' error
11.   --  INSERT INTO stats (symbol, ratio) VALUES ('XYZ', NULL);
12.   -- COMMIT;
13.
14.   WHEN OTHERS THEN  -- handles all other errors
15.   p_result:='failure';
16.
17.   END ACCOUNT_INSERT_PROC;
18.   /
```

Before executing of this application Account table.

ACCNO	NAME	BAL
1001	sekhar	6800

After executing application, Account table is:

ACCNO	NAME	BAL
1001	somasekhar	6800
1002	yerragudi	6800
1003	l.n.rao	5600
1004	kesavareddy	5600
1005	yerragudi	4600

OUTPUT

```
----usingConnectionCallback----

No.of rows updated : 1

----usingStatementCallback----
Account Details are ....
1001
somasekhar
6800.0

----usingPreparedStatementCallback----
No.of rows inserted : 1

----usingPreparedStatementCallback----
No.of rows inserted : 1

----usingPreparedStatementCallback003----
No.of rows inserted : 1
----usingCallableStatementCallback----
Procedure execution status :success
----usingPreparedStatementCreator-----
1001
somasekhar
6800.0
1002
yerragudi
6800.0
1003
l.n.rao
5600.0
1004
kesavareddy
5600.0
1005
yerragudi
4600.0
```

Q) How to implement DML statement (insert, update, delete) in spring framework?

Ans: JdbcTEmplate is providing overloaded **update()** methods to implement DML statements.

public int update(String sql):-

When the DML query is not having any placeholders then we will go for this method.

Ex:- DELETE FROM ACCOUNT WHERE ACCNO=1001;

public int update(String sql, Object[] args):-

When the DML query is having placeholders we will use this method.

Ex:- String query= "INSERT INTO ACCOUNT VALUES(?,?,?)";

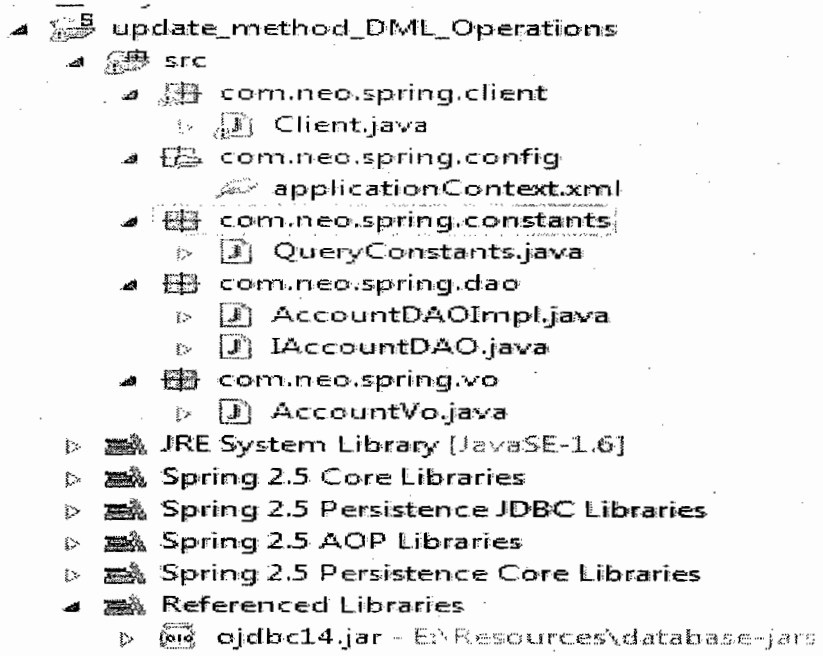
jdbcTemplate.update(query, new Object[]{account.getAccno(), account.getName(), account.getBal()});

public int update(String sql, Object[] args, int[] args):-

When the query is having place holders & if we want to specify place holder types then we will use this method.

Ex:- String query = "UPDATE ACCOUNT SET NAME=?, BAL=? WHERE ACCNO=?;

jdbcTemplate.update(query, new Object[]{account.getName(), account.getBal(), account.getAccno()}, new int[]{Types.VARCHAR, Types.DOUBLE, Types.INTEGER});



applicationContext.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3.     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xmlns:p="http://www.springframework.org/schema/p"
5.     xsi:schemaLocation="http://www.springframework.org/schema/beans
6. http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
```

```
7.
8. <!--  Apache datasource configuration -->
9.     <bean id="apacheDatasourceRef"
10.                          class="org.apache.commons.dbcp.BasicDataSource">
11.      <property name="driverClassName"
12.                                  value="oracle.jdbc.driver.OracleDriver" />
13.      <property name="url" value="jdbc:oracle:thin:@localhost:1521:XE" />
14.      <property name="username" value="system" />
15.      <property name="password" value="tiger" />
16.    </bean>
17.
18.    <bean id="jdbcTemplateRef"
19.                        class="org.springframework.jdbc.core.JdbcTemplate" >
20.          <property name="dataSource" ref="apacheDatasourceRef" />
21.    </bean>
22.
23.    <bean id="accountDAORef" class="com.neo.spring.dao.AccountDAOImpl">
24.          <property name="dataSource" ref="jdbcTemplateRef"></property>
25.    </bean>
26.
27.    </beans>
```

QueryConstants.java

```
1. package com.neo.spring.constants;
2.
3. public interface QueryConstants {
4.
5.     public static String INSERT_QUERY_001 = "INSERT INTO ACCOUNT
6.                                             VALUES(1002,'sekhar',3500.0)";
7.     public static String INSERT_QUERY_002 = "INSERT INTO ACCOUNT
8.                                                       VALUES(?,?,?)";
9.     public static String UPDATE_QUERY = "UPDATE ACCOUNT SET NAME=?, BAL=?
10.                                                       WHERE ACCNO=?";
11.    public static String DELETE_QUERY = "DELETE  FROM ACCOUNT WHERE
12.                                                             ACCNO=?";
13.
14.    }
```

AccountVo.java

```
1. package com.neo.spring.vo;
2.
3. public class AccountVo {
4.     private int accno;
5.     private String name;
6.     private double balance;
7.
8.     public int getAccno() {
9.           return accno;
10.          }
11.
12.          public void setAccno(int accno) {
13.                this.accno = accno;
14.          }
15.
16.          public String getName() {
```

```
17.               return name;
18.         }
19.
20.         public void setName(String name) {
21.               this.name = name;
22.         }
23.
24.         public double getBalance() {
25.               return balance;
26.         }
27.
28.         public void setBalance(double balance) {
29.               this.balance = balance;
30.         }
31.
32.   }
```

IAccountDAO.java

```
1. package com.neo.spring.dao;
2.
3. import com.neo.spring.vo.AccountVo;
4.
5. public interface IAccountDAO {
6.     public int insert001();
7.     public int insert002(AccountVo accountVo);
8.     public int insert003(AccountVo accountVo);
9.     public int update(AccountVo accountVo);
10.          public int delete(int accno);
11.
12.
13.   }
```

AccountDAOImpl.java

```
1. package com.neo.spring.dao;
2.
3. import java.sql.Types;
4.
5. import org.springframework.jdbc.core.JdbcTemplate;
6.
7. import com.neo.spring.constants.QueryConstants;
8. import com.neo.spring.vo.AccountVo;
9.
10.   public class AccountDAOImpl implements IAccountDAO {
11.
12.         @Override
13.         public int insert001() {
14.               return template.update(QueryConstants.INSERT_QUERY_001);
15.         }
16.
17.         @Override
18.         public int insert002(AccountVo accountVo) {
19.   //    Object[] objects = new Object[]{accountVo.getAccno(),
20.                               accountVo.getName(), accountVo.getBalance()};
21.   //    template.update(QueryConstants.INSERT_QUERY_002, objects);
22.
```

```
23.         return template.update(QueryConstants.INSERT_QUERY_002,
24.                   new Object[]{accountVo.getAccno(), accountVo.getName(),
25.                                                   accountVo.getBalance()});
26.         }
27.
28.         @Override
29.         public int insert003(AccountVo accountVo) {
30.               return template.update(QueryConstants.INSERT_QUERY_002,
31.                  new Object[]{accountVo.getAccno(), accountVo.getName(),
32.                  accountVo.getBalance()}, new int[]{Types.INTEGER,
33.                                          Types.VARCHAR, Types.DOUBLE});
34.         }
35.
36.         @Override
37.         public int update(AccountVo accountVo) {
38.               return template.update(QueryConstants.UPDATE_QUERY,
39.                     new Object[]{accountVo.getName(),
40.                     accountVo.getBalance(),accountVo.getAccno()}, new
41.                     int[]{ Types.VARCHAR, Types.DOUBLE, Types.INTEGER});
42.         }
43.
44.         @Override
45.         public int delete(int accno) {
46.               return template.update(QueryConstants.DELETE_QUERY, new
47.                              Object[]{accno}, new int[]{Types.INTEGER});
48.         }
49.
50.         private JdbcTemplate template;
51.
52.         public void setTemplate(JdbcTemplate template) {
53.               this.template = template;
54.         }
55.   }
```

Client.java

```
1. package com.neo.spring.client;
2.
3. import org.springframework.context.ApplicationContext;
4. import org.springframework.context.support.ClassPathXmlApplicationContext;
5.
6. import com.neo.spring.dao.IAccountDAO;
7. import com.neo.spring.vo.AccountVo;
8.
9. public class Client {
10.         private static ApplicationContext context = new
11.                 ClassPathXmlApplicationContext(
12.                     "com/neo/spring/config/applicationContext.xml");
13.
14.         public static void main(String[] args) {
15.               IAccountDAO accountDAO = (IAccountDAO)
16.                                    context.getBean("accountDAORef");
17.               sop("----update(String query)----");
18.               int ra = accountDAO.insert001();
19.               sop("No.of rows inserted : "+ra);
20.
```

```
21.                 sop("----update(String query, Object[] args)----");
22.                 AccountVo accountVoA = new AccountVo();
23.                 accountVoA.setAccno(1006);
24.                 accountVoA.setName("yellareddy");
25.                 accountVoA.setBalance(4700.0);
26.                 ra = accountDAO.insert002(accountVoA);
27.                 sop("No.of rows inserted : "+ra);
28.
29.                 AccountVo accountVoB = new AccountVo();
30.                 accountVoB.setAccno(1007);
31.                 accountVoB.setName("somu");
32.                 accountVoB.setBalance(9800.0);
33.                 ra = accountDAO.insert003(accountVoB);
34.                 sop("NO.of rows inserted : "+ra);
35.
36.                 AccountVo accountVoC = new AccountVo();
37.                 accountVoC.setAccno(1007);
38.                 accountVoC.setName("somasekhar");
39.                 accountVoC.setBalance(8689.0);
40.                 ra = accountDAO.update(accountVoC);
41.                 sop("NO.of rows updated : "+ra);
42.
43.                 ra = accountDAO.delete(1005);
44.                 sop("No.of rows deleted : "+ra);
45.         }
46.         public static void sop(Object object){
47.                 System.out.println(object);
48.         }
49.   }
```

Before executing of this application Account table.

ACCNO	NAME	BAL
1004	kesavareddy	5600

After executing application, Account table is:

ACCNO	NAME	BAL
1002	yellareddy	4700
1003	somasekhar	8689
1002	sekhar	3500

OUTPUT

```
----update(String query)----
No.of rows inserted : 1
----update(String query, Object[] args)----
No.of rows inserted : 1
----update(String query, Object[] args, int[] types)----
NO.of rows inserted : 1
----update(String query)----
NO.of rows updated : 1
----update(String query)----
No.of rows deleted : 1
```

Q) What is the need of AbstractDao?

Ans: Let us consider the following beans:

```
public class EmployeeDaoImpl.....{
    private JdbcTemplate jdbcTemplate;
    public void setJt(JdbcTemplate jdbcTemplate){
       this. jdbcTemplate= jdbcTemplate;
       }
   ------------------
   ------------------
  //employee specific dao methods
}
```

```
public class AccountDaoImpl.....{
    private JdbcTemplate  jdbcTemplate ;
    public void setJt(JdbcTemplate  jdbcTemplate){
          this. jdbcTemplate= jdbcTemplate;
       }
   ------------------
   ------------------
  //account specific dao methods
}
```

If we observe the above DAO classes, the **JdbcTemplate injectioin logic** is repeating.

So, why don't we have a class which is having the above common logic (JdbcTemplate injection logic) and why don't we extend that one to all DAO classes.

Let's write the class as below:

```
public class AbstractDao{
       private JdbcTemplate jdbcTemplate;
       public void setJdbcTemplate (JdbcTemplate jdbcTemplate){
       this. jdbcTemplate= jdbcTemplate;
       }
      public JdbcTemplate  getJdbcTemplate (){
        return jdbcTemplate;
      }
}
```

If anyone creates the object of AbstractDao we can't get any services from AbstractDao object, because it just contains JdbcTemplate injects code.

So, it is better to make the above class as abstract class.

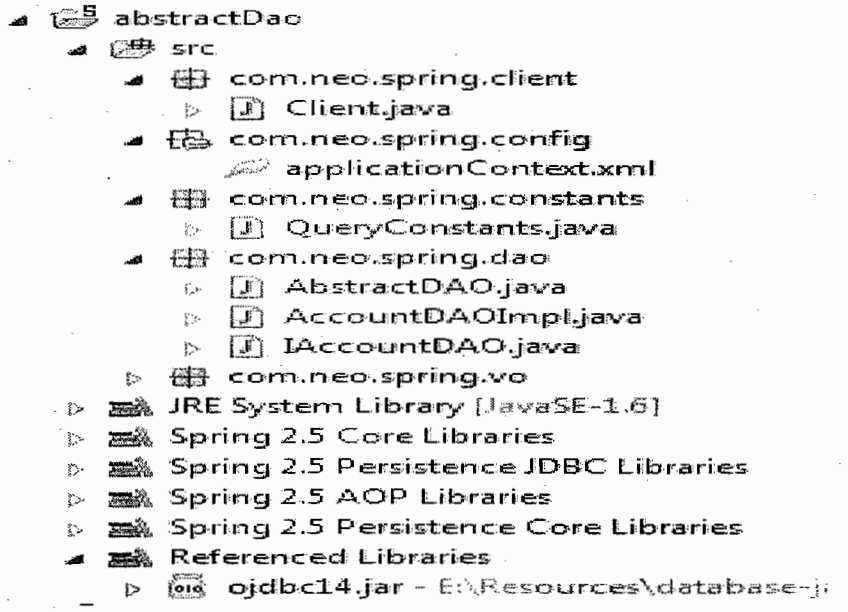
AbstractDao.java

```
public abstract class AbstractDao{
       private JdbcTemplate jdbcTemplate;
       public void setJdbcTemplate (JdbcTemplate jdbcTemplate){
       this. jdbcTemplate= jdbcTemplate;
       }
      public JdbcTemplate  getJdbcTemplate (){
        return jdbcTemplate;
      }
}
```

AccountDaoImpl.java

```
public class AccountDaoImpl extends AbstractDao{
        ----------
        ----------
      public void insert(Account account){
         getJdbcTemplate().update(query.....);
         ----------
         ----------
         // account specific dao methods
         }
}
```

Here we are calling the method to **getJdbcTemplate()** method, and this method returns **JdbcTemplate** object. Actually this method is available in **AbstractDao** class.



applicationContext.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3.     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xmlns:p="http://www.springframework.org/schema/p"
5.     xsi:schemaLocation="http://www.springframework.org/schema/beans
6. http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
7.
8. <!--  Apache datasource configuration -->
9.     <bean id="apacheDatasourceRef"
10.                          class="org.apache.commons.dbcp.BasicDataSource">
11.      <property name="driverClassName"
12.                                  value="oracle.jdbc.driver.OracleDriver" />
13.      <property name="url" value="jdbc:oracle:thin:@localhost:1521:XE" />
14.      <property name="username" value="system" />
```

```
15.     <property name="password" value="tiger" />
16.   </bean>
17.
18.   <bean id="jdbcTemplateRef"
19.                       class="org.springframework.jdbc.core.JdbcTemplate" >
20.         <property name="dataSource" ref="apacheDatasourceRef" />
21.   </bean>
22.
23.   <bean id="accountDAORef" class="com.neo.spring.dao.AccountDAOImpl">
24.         <property name="dataSource" ref="jdbcTemplateRef"></property>
25.   </bean>
26.
27.   </beans>
```

QueryConstants.java

```
1. package com.neo.spring.constants;
2.
3. public interface QueryConstants {
4.    public static String UPDATE_QUERY = "UPDATE ACCOUNT SET NAME=?, BAL=?
5.                                                          WHERE ACCNO=?";
6. }
```

AccountVo.java

```
1. package com.neo.spring.vo;
2.
3. public class AccountVo {
4.    private int accno;
5.    private String name;
6.    private double balance;
7.
8.    public AccountVo(int accno, String name, double balance) {
9.          this.accno = accno;
10.               this.name = name;
11.               this.balance = balance;
12.         }
13.         public int getAccno() {
14.               return accno;
15.         }
16.
17.         public void setAccno(int accno) {
18.               this.accno = accno;
19.         }
20.
21.         public String getName() {
22.               return name;
23.         }
24.
25.         public void setName(String name) {
26.               this.name = name;
27.         }
28.
29.         public double getBalance() {
30.               return balance;
31.         }
32.
```

```
33.         public void setBalance(double balance) {
34.               this.balance = balance;
35.         }
36.
37.   }
```

IAccountDAO.java

```
1. package com.neo.spring.dao;
2.
3. import com.neo.spring.vo.AccountVo;
4.
5. public interface IAccountDAO {
6.    public int update(AccountVo accountVo);
7. }
```

AbstractDAO.java

```
1. package com.neo.spring.dao;
2.
3. import org.springframework.jdbc.core.JdbcTemplate;
4.
5. public abstract class AbstractDAO {
6.    private JdbcTemplate jdbcTemplate;
7.
8.    public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
9.          this.jdbcTemplate = jdbcTemplate;
10.         }
11.
12.         public JdbcTemplate getJdbcTemplate() {
13.               return jdbcTemplate;
14.         }
15.
16.   }
```

AccountDAOImpl.java

```
1. package com.neo.spring.dao;
2.
3. import java.sql.Types;
4.
5. import com.neo.spring.constants.QueryConstants;
6. import com.neo.spring.vo.AccountVo;
7.
8. public class AccountDAOImpl extends AbstractDAO implements IAccountDAO {
9.          @Override
10.         public int update(AccountVo accountVo) {
11.               return getJdbcTemplate().update(QueryConstants.UPDATE_QUERY
12.                     , new Object[]{accountVo.getName(),
13.                      accountVo.getBalance(), accountVo.getAccno()}, new
14.                      int[]{Types.VARCHAR, Types.DOUBLE, Types.INTEGER});
15.         }
16.   }
```

Client.java

```
1. package com.neo.spring.client;
2.
3. import org.springframework.context.ApplicationContext;
```

```
4. import org.springframework.context.support.ClassPathXmlApplicationContext;
5.
6. import com.neo.spring.dao.IAccountDAO;
7. import com.neo.spring.vo.AccountVo;
8.
9. public class Client {
10.         private static ApplicationContext context = new
11.                 ClassPathXmlApplicationContext(
12.                       "com/neo/spring/config/applicationContext.xml");
13.
14.         public static void main(String[] args) {
15.               IAccountDAO accountDAO = (IAccountDAO)
16.                                       context.getBean("accountDAORef");
17.               AccountVo accountVo = new AccountVo(1001, "sekharreddy",
18.                                                              2490.0);
19.               int ra = accountDAO.update(accountVo);
20.               System.out.println("No.of rows updated : "+ra);
21.   }
22.   }
```

NOTE : In the above example we are inject JdbcTemplate object to AccountDAOImpl class. Instead of that we can directly inject DataSource to AccountDAOImpl, So that we can eliminate the JdbcTemplate object configuration in spring configuration file.

Q.) Rewrite the AbstractDAO class, spring configuration file so that we can inject DataSource to AccountDAOImpl ?

AbstractDAO.java

```
1. package com.neo.spring.dao;
2.
3. import javax.sql.DataSource;
4.
5. import org.springframework.jdbc.core.JdbcTemplate;
6.
7. public abstract class AbstractDAO {
8.    private JdbcTemplate jdbcTemplate;
9.
10.         public void setDataSource(DataSource dataSource) {
11.               jdbcTemplate = new JdbcTemplate(dataSource);
12.         }
13.
14.         public JdbcTemplate getJdbcTemplate() {
15.               return jdbcTemplate;
16.         }
17.
18.   }
```

applicationContext.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3.    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xmlns:p="http://www.springframework.org/schema/p"
5.    xsi:schemaLocation="http://www.springframework.org/schema/beans
6. http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
```

```
7.
8. <!--  Apache datasource configuration -->
9.     <bean id="apacheDatasourceRef"
10.                          class="org.apache.commons.dbcp.BasicDataSource">
11.      <property name="driverClassName"
12.                                  value="oracle.jdbc.driver.OracleDriver" />
13.      <property name="url" value="jdbc:oracle:thin:@localhost:1521:XE" />
14.      <property name="username" value="system" />
15.      <property name="password" value="tiger" />
16.    </bean>
17.
18.    <bean id="accountDAORef" class="com.neo.spring.dao.AccountDAOImpl">
19.          <property name="dataSource" ref=" apacheDatasourceRef " />
20.    </bean>
21.
22.    </beans>
```

Q) What is the need of JdbcDaoSupport?

- It is similar to **AbstractDao** class and which is given by SpringFramework.
- So instead of we developing the **AbstractDao**, we can use the spring given **JdbcDaoSupport**.

Example :

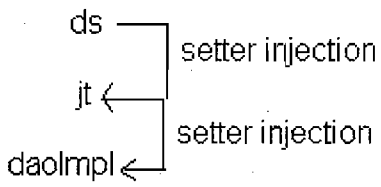
```
public class AccountDaoImpl extends JdbcDaoSupport{
    ----------
    ----------
    public void insert(Account account){
        getJdbcTemplate().update(query.....);
        ----------
        ----------
        // account specific dao methods
        }
}
```

Here we are calling the method to **getJdbcTemplate()** method to get **JdbcTemplate** object. Actually this method is available in **JdbcDaoSupport** class.

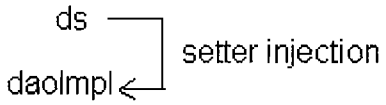
Q) What is the flexibility given by JdbcDaoSupport to configure Dao classes in spring configuration file?

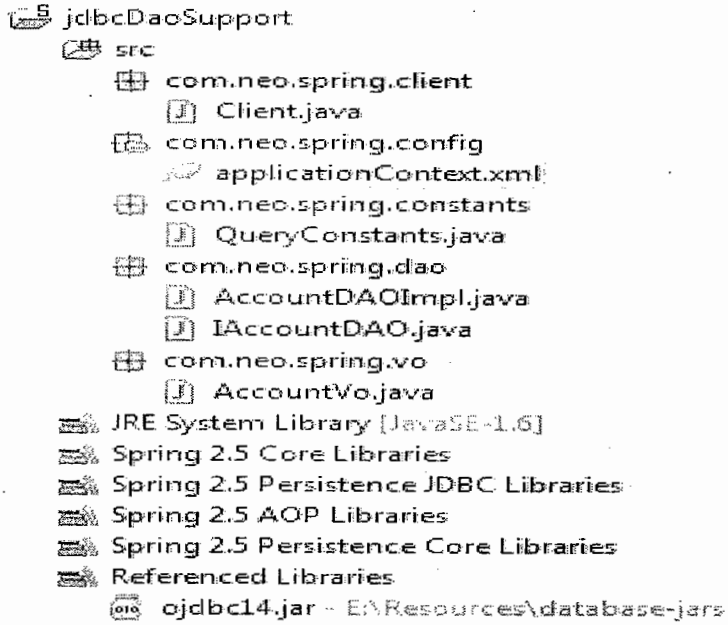
Ans: We can directly inject **DataSource** object (in the constructor approach) to DaoImpl objects, instead of injecting **JdbcTemplate** object.

Without JdbcDaoSupport in spring configuration



With JdbcDaoSupport in spring configuration





applicationContext.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3.     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xmlns:p="http://www.springframework.org/schema/p"
5.     xsi:schemaLocation="http://www.springframework.org/schema/beans
6. http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
7.
8. <!--  Apache datasource configuration -->
9.     <bean id="apacheDatasourceRef"
10.                          class="org.apache.commons.dbcp.BasicDataSource">
11.      <property name="driverClassName"
12.                                  value="oracle.jdbc.driver.OracleDriver" />
13.      <property name="url" value="jdbc:oracle:thin:@localhost:1521:XE" />
14.      <property name="username" value="system" />
15.      <property name="password" value="tiger" />
16.    </bean>
17.
18.    <bean id="accountDAORef" class="com.neo.spring.dao.AccountDAOImpl">
19.          <property name="dataSource" ref=" apacheDatasourceRef " />
20.    </bean>
21.
22.    </beans>
```

QueryConstants.java

```
1. package com.neo.spring.constants;
2.
3. public interface QueryConstants {
4.     public static String INSERT_QUERY = "INSERT INTO ACCOUNT
5.                                                    VALUES(?,?,?)";
6.     public static String UPDATE_QUERY = "UPDATE ACCOUNT SET NAME=?, BAL=?
7.                                                     WHERE ACCNO=?";
8.     public static String DELETE_QUERY = "DELETE  FROM ACCOUNT WHERE
9.                                                         ACCNO=?";
10.    }
```

AccountVo.java

```
1. package com.neo.spring.vo;
2.
3. public class AccountVo {
4.     private int accno;
5.     private String name;
6.     private double balance;
7.
8.     public AccountVo(int accno, String name, double balance) {
9.           this.accno = accno;
10.                this.name = name;
11.                this.balance = balance;
12.          }
13.          //setters & getters
14.
15.    }
```

IAccountDAO.java

```
1. package com.neo.spring.dao;
2. import com.neo.spring.vo.AccountVo;
3. public interface IAccountDAO {
4.     public int insert(AccountVo accountVo);
5.     public int update(AccountVo accountVo);
6.     public int delete(int accno);
7.
8. }
```

AccountDAOImpl.java

```
1. package com.neo.spring.dao;
2. import java.sql.Types;
3. import org.springframework.jdbc.core.support.JdbcDaoSupport;
4. import com.neo.spring.constants.QueryConstants;
5. import com.neo.spring.vo.AccountVo;
6. public class AccountDAOImpl extends JdbcDaoSupport
7.                                                  implements IAccountDAO {
8.
9. @Override
10.    public int insert(AccountVo accountVo) {
11.          return getJdbcTemplate().update(QueryConstants.INSERT_QUERY, new
12.                      Object[]{accountVo.getAccno(), accountVo.getName(),
13.                      accountVo.getBalance()}, new int[]{Types.INTEGER,
14.                      Types.VARCHAR, Types.DOUBLE});
15.    }
16.
17.    @Override
18.    public int update(AccountVo accountVo) {
19.          return getJdbcTemplate().update(QueryConstants.UPDATE_QUERY, new
20.                    Object[]{accountVo.getName(),accountVo.getBalance(),
21.                    accountVo.getAccno()}, new int[]{ Types.VARCHAR,
22.                    Types.DOUBLE, Types.INTEGER});
23.    }
24.
25.    @Override
26.    public int delete(int accno) {
```

```
27.         return getJdbcTemplate().update(QueryConstants.DELETE_QUERY, new
28.                                 Object[]{accno}, new int[]{Types.INTEGER});
29.    }
30.    }
```

Client.java

```
1. package com.neo.spring.client;
2.
3. import org.springframework.context.ApplicationContext;
4. import org.springframework.context.support.ClassPathXmlApplicationContext;
5. import com.neo.spring.dao.IAccountDAO;
6. import com.neo.spring.vo.AccountVo;
7. public class Client {
8. private static ApplicationContext context = new
9.           ClassPathXmlApplicationContext(
10.                    "com/neo/spring/config/applicationContext.xml");
11.
12.   public static void main(String[] args) {
13.         IAccountDAO accountDAO = (IAccountDAO)
14.                                       context.getBean("accountDAORef");
15.
16.         sop("---insert---");
17.         AccountVo accountVoA =new AccountVo(1006, "sekharreddy", 2490.0);
18.         int ra = accountDAO.insert(accountVoA);
19.         sop("NO.of Rows inserted : " + ra);
20.
21.         sop("---update---");
22.         AccountVo accountVoB = new AccountVo(1001, "somasekhar", 9490.0);
23.         ra = accountDAO.update(accountVoB);
24.         sop("NO.of rows updated : " + ra);
25.
26.         sop("---delete---");
27.         ra = accountDAO.delete(1001);
28.         sop("NO.of rows deleted : " + ra);
29.
30.   }
31.
32.   public static void sop(Object object) {
33.         System.out.println(object);
34.   }
35.   }
```

Before execution Account table

ACCNO	NAME	BAL
1001	sekhar	8970
1002	yellareddy	4500

After execution Account table

ACCNO	NAME	BAL
1001	somasekhar	9490
1005	sekharreddy	2490

OUTPUT

```
---insert---
NO.of Rows inserted : 1
---update---
NO.of rows updated : 1
---delete---
NO.of rows deleted : 1
```

Q) What is SimpleJdbcTemplate?

Ans: It is very similar to **JdbcTemplate** object but it's using the varargs feature of jdk 1.5 and named parameters.

With JdbcTemplate

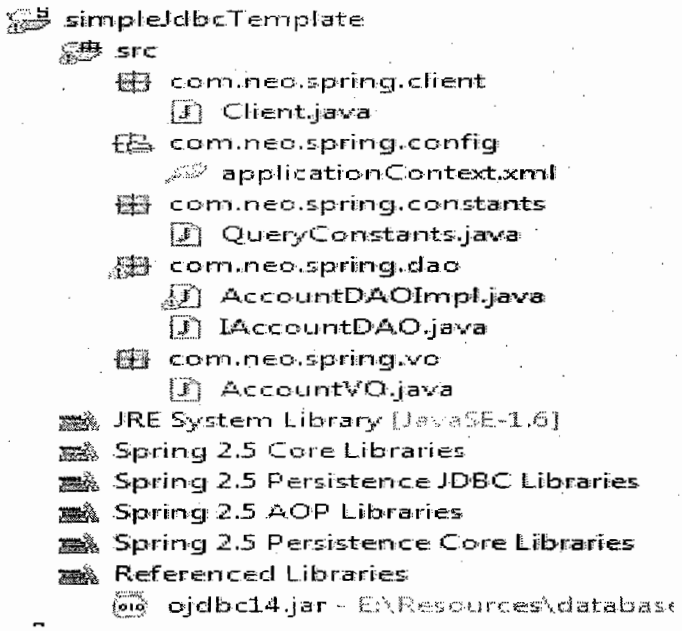
Ex:- String query ="INSERT INTO ACCOUNT VALUES(?,?,?);

jdbcTemplate.update(query, new Object[]{account.getAccno(), account.getName(), account.getBal()});

With SimpleJdbcTemplate

simpleJdbcTemplate.update(query, account.getAccno(), account.getName(), account.getBal());

SimpleJdbcTemplate does not provide all the methods of JdbcTemplate as varargs concept is having some restrictions.



applicationContext.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3.     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xmlns:p="http://www.springframework.org/schema/p"
5.     xsi:schemaLocation="http://www.springframework.org/schema/beans
6. http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
7.
8. <!--  Apache datasource configuration -->
9.     <bean id="apacheDatasourceRef"
10.                          class="org.apache.commons.dbcp.BasicDataSource">
11.      <property name="driverClassName"
12.                                value="oracle.jdbc.driver.OracleDriver" />
13.      <property name="url" value="jdbc:oracle:thin:@localhost:1521:XE" />
14.      <property name="username" value="system" />
15.      <property name="password" value="tiger" />
16.    </bean>
17.
18.    <bean id="simpleJTRef"
```

```
19.         class="org.springframework.jdbc.core.simple.SimpleJdbcTemplate">
20.         <constructor-arg ref="apacheDatasourceRef"></constructor-arg>
21.   </bean>
22.
23.   <bean id="accountDAORef" class="com.neo.spring.dao.AccountDAOImpl">
24.         <property name="simpleJdbcTemplate" ref="simpleJTRef"></property>
25.   </bean>
26.
27.   </beans>
```

QueryConstants.java

```
1. package com.neo.spring.constants;
2.
3. public interface QueryConstants {
4.    public static String INSERT_QUERY = "INSERT INTO ACCOUNT
5.                                                    VALUES(?,?,?)";
6.    public static String UPDATE_QUERY = "UPDATE ACCOUNT SET NAME=?, BAL=?
7.                                                     WHERE ACCNO=?";
8.    public static String DELETE_QUERY = "DELETE  FROM ACCOUNT WHERE
9.                                                          ACCNO=?";
10.   }
```

AccountVO.java

```
1. package com.neo.spring.vo;
2.
3. public class AccountVO{
4.    private int accno;
5.    private String name;
6.    private double balance;
7.
8.    public AccountVO(int accno, String name, double balance) {
9.          this.accno = accno;
10.               this.name = name;
11.               this.balance = balance;
12.         }
13.         //setters & getters
14.
15.   }
```

IAccountDAO.java

```
1. package com.neo.spring.dao;
2.
3. import com.neo.spring.vo.AccountVO;
4.
5. public interface IAccountDAO {
6.    public int insert(AccountVO accountVO);
7.    public int update(AccountVO accountVO);
8.    public int delete(int accno);
9.
10.   }
```

AccountDAOImpl.java

```
1. package com.neo.spring.dao;
2.
3. import org.springframework.jdbc.core.simple.SimpleJdbcTemplate;
```

```
4.
5. import com.neo.spring.constants.QueryConstants;
6. import com.neo.spring.vo.AccountVO;
7.
8. public class AccountDAOImpl implements IAccountDAO {
9.
10.   @Override
11.   public int insert(AccountVO accountVo) {
12.      return simpleJdbcTemplate.update(QueryConstants.INSERT_QUERY,
13.       accountVo.getAccno(), accountVo.getName(), accountVo.getBalance());
14.   }
15.
16.   @Override
17.   public int update(AccountVO accountVo) {
18.    return simpleJdbcTemplate.update(QueryConstants.UPDATE_QUERY,
19.     accountVo.getName(), accountVo.getBalance(), accountVo.getAccno());
20.   }
21.
22.   @Override
23.   public int delete(int accno) {
24.     return simpleJdbcTemplate.update(QueryConstants.DELETE_QUERY, accno);
25.   }
26.
27.   private SimpleJdbcTemplate simpleJdbcTemplate;
28.
29.   public void setSimpleJdbcTemplate(SimpleJdbcTemplate
30.                                          simpleJdbcTemplate) {
31.         this.simpleJdbcTemplate = simpleJdbcTemplate;
32.   }
33.
34.   }
```

Client.java

```
1. package com.neo.spring.client;
2.
3. import org.springframework.context.ApplicationContext;
4. import org.springframework.context.support.ClassPathXmlApplicationContext;
5.
6. import com.neo.spring.dao.IAccountDAO;
7. import com.neo.spring.vo.AccountVo;
8.
9. public class Client {
10.   private static ApplicationContext context = new
11.             ClassPathXmlApplicationContext(
12.                    "com/neo/spring/config/applicationContext.xml");
13.
14.   public static void main(String[] args) {
15.         IAccountDAO accountDAO = (IAccountDAO)
16.                                       context.getBean("accountDAORef");
17.
18.         sop("---insert---");
19.         AccountVO accountVoA =new AccountVO(1006, "sekharreddy", 2490.0);
20.         int ra = accountDAO.insert(accountVoA);
21.         sop("NO.of Rows inserted : " + ra);
22.
```

```
23.         sop("---update---");
24.         AccountVO accountVoB = new AccountVO(1001, "somasekhar", 9490.0);
25.         ra = accountDAO.update(accountVoB);
26.         sop("NO.of rows updated : " + ra);
27.
28.         sop("---delete---");
29.         ra = accountDAO.delete(1001);
30.         sop("NO.of rows deleted : " + ra);
31.
32.   }
33.
34.   public static void sop(Object object) {
35.         System.out.println(object);
36.   }
37.
38.   }
```

Before execution Account table

ACCNO	NAME	BAL
1001	sekhar	8970
1002	yellareddy	4500

After execution Account table

ACCNO	NAME	BAL
1001	somasekhar	9490
1005	sekharreddy	2490

OUTPUT

```
---insert---
NO.of Rows inserted : 1
---update---
NO.of rows updated : 1
---delete---
NO.of rows deleted : 1
```

Q) What is SimpleJdbcDaoSupport?

- It is very similar to **JdbcDaoSupport**.
- Instead of we writing injection logic (**SimpleJdbcTemplate** injection) i.e., already provided by **SimpleJdbcDaoSupport**.
- So every Dao class (which needs **SimpleJdbcTemplate** better to extend **SimpleJdbcDaoSupport**).

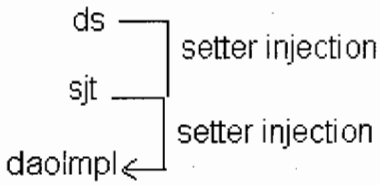
```
public class AccountDaoImpl extends SimpleJdbcDaoSupport{
    public void insert(Account account){
        getSimpleJdbcTemplate().update(query.....);
        ----------
        ----------
          // account specific dao methods
    }
}
```

Here we are calling the method **getSimpleJdbcTemplate()** to get **SimpleJdbcTemplate** object. This method is available in **SimpleJdbcDaoSupport** class.

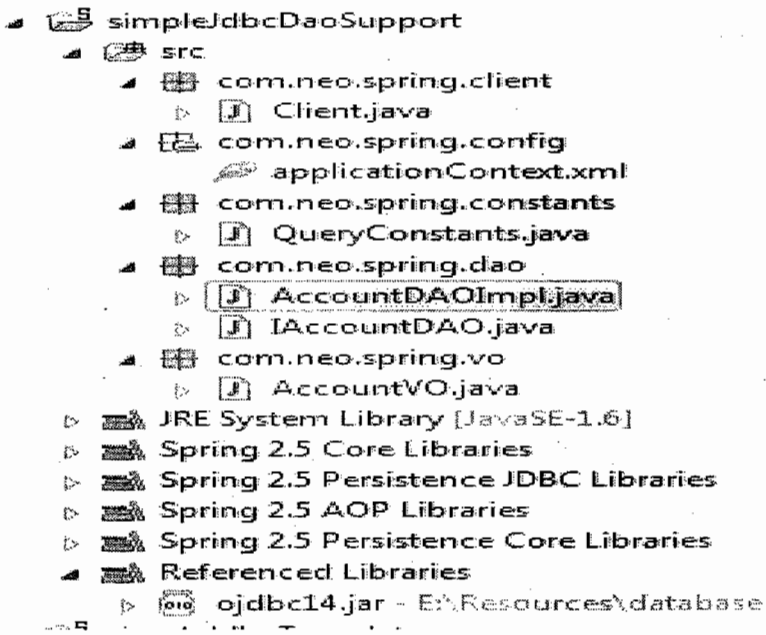
Q) What is the flexibility given by SimpleJdbcDaoSupport to configure Dao classes in spring configuration file?

Ans: Instead of injecting **SimpleJdbcTemplate** object into the Dao class, we can inject **DataSource** in the constructor approach.

Without SimpleJdbcDaoSupport in spring configuration



With SimpleJdbcDaoSupport in spring configuration



applicationContext.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3.     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xmlns:p="http://www.springframework.org/schema/p"
5.     xsi:schemaLocation="http://www.springframework.org/schema/beans
6. http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
7.
8. <!--  Apache datasource configuration -->
9.     <bean id="apacheDatasourceRef"
10.                          class="org.apache.commons.dbcp.BasicDataSource">
11.      <property name="driverClassName"
12.                                  value="oracle.jdbc.driver.OracleDriver" />
13.      <property name="url" value="jdbc:oracle:thin:@localhost:1521:XE" />
14.      <property name="username" value="system" />
15.      <property name="password" value="tiger" />
16.    </bean>
17.
18.    <bean id="accountDAORef" class="com.neo.spring.dao.AccountDAOImpl">
19.        <property name="dataSource" ref="apacheDatasourceRef"></property>
20.    </bean>
21.
22.    </beans>
```

QueryConstants.java

```
1. package com.neo.spring.constants;
2.
3. public interface QueryConstants {
4.     public static String INSERT_QUERY = "INSERT INTO ACCOUNT
5.                                                    VALUES(?,?,?)";
6.     public static String UPDATE_QUERY = "UPDATE ACCOUNT SET NAME=?, BAL=?
7.                                                    WHERE ACCNO=?";
8.     public static String DELETE_QUERY = "DELETE  FROM ACCOUNT WHERE
9.                                                    ACCNO=?";
10.    }
```

AccountVO.java

```
1. package com.neo.spring.vo;
2.
3. public class AccountVO{
4.     private int accno;
5.     private String name;
6.     private double balance;
7.
8.     public AccountVO(int accno, String name, double balance) {
9.           this.accno = accno;
10.                this.name = name;
11.                this.balance = balance;
12.          }
13.          //setters & getters
14.
15.    }
```

IAccountDAO.java

```
1. package com.neo.spring.dao;
2.
3. import com.neo.spring.vo.AccountVO;
4.
5. public interface IAccountDAO {
6.     public int insert(AccountVO accountVO);
7.     public int update(AccountVO accountVO);
8.     public int delete(int accno);
9.
10.    }
```

AccountDAOImpl.java

```
1. package com.neo.spring.dao;
2.
3. import org.springframework.jdbc.core.simple.SimpleJdbcDaoSupport;
4.
5. import com.neo.spring.constants.QueryConstants;
6. import com.neo.spring.vo.AccountVO;
7.
8. public class AccountDAOImpl extends SimpleJdbcDaoSupport
9.                                          implements IAccountDAO {
10.
11.    @Override
12.    public int insert(AccountVO accountVo) {
13.          return getSimpleJdbcTemplate().update(
14.                QueryConstants.INSERT_QUERY, accountVo.getAccno(),
15.                      accountVo.getName(), accountVo.getBalance());
16.    }
17.
18.    @Override
19.    public int update(AccountVO accountVo) {
20.          return getSimpleJdbcTemplate().update(
21.                  QueryConstants.UPDATE_QUERY, accountVo.getName(),
22.                      accountVo.getBalance(), accountVo.getAccno());
23.    }
24.
25.    @Override
26.    public int delete(int accno) {
27.          return getSimpleJdbcTemplate().update(
28.                                 QueryConstants.DELETE_QUERY, accno);
29.    }
30.
31.    }
```

Client.java

```
1. package com.neo.spring.client;
2.
3. import org.springframework.context.ApplicationContext;
4. import org.springframework.context.support.ClassPathXmlApplicationContext;
5.
6. import com.neo.spring.dao.IAccountDAO;
7. import com.neo.spring.vo.AccountVo;
8.
9. public class Client {
```

```
10.   private static ApplicationContext context = new
11.             ClassPathXmlApplicationContext(
12.                     "com/neo/spring/config/applicationContext.xml");
13.
14.   public static void main(String[] args) {
15.         IAccountDAO accountDAO = (IAccountDAO)
16.                                         context.getBean("accountDAORef");
17.
18.         sop("---insert---");
19.         AccountVO accountVoA =new AccountVO(1006, "sekharreddy", 2490.0);
20.         int ra = accountDAO.insert(accountVoA);
21.         sop("NO.of Rows inserted : " + ra);
22.
23.         sop("---update---");
24.         AccountVO accountVoB = new AccountVO(1001, "somasekhar", 9490.0);
25.         ra = accountDAO.update(accountVoB);
26.         sop("NO.of rows updated : " + ra);
27.
28.         sop("---delete---");
29.         ra = accountDAO.delete(1001);
30.         sop("NO.of rows deleted : " + ra);
31.
32.   }
33.
34.   public static void sop(Object object) {
35.         System.out.println(object);
36.   }
37.
38.   }
```

Before execution Account table

ACCNO	NAME	BAL
1001	sekhar	8970
1002	yellareddy	4500

After execution Account table

ACCNO	NAME	BAL
1001	somasekhar	9490
1005	sekharreddy	2490

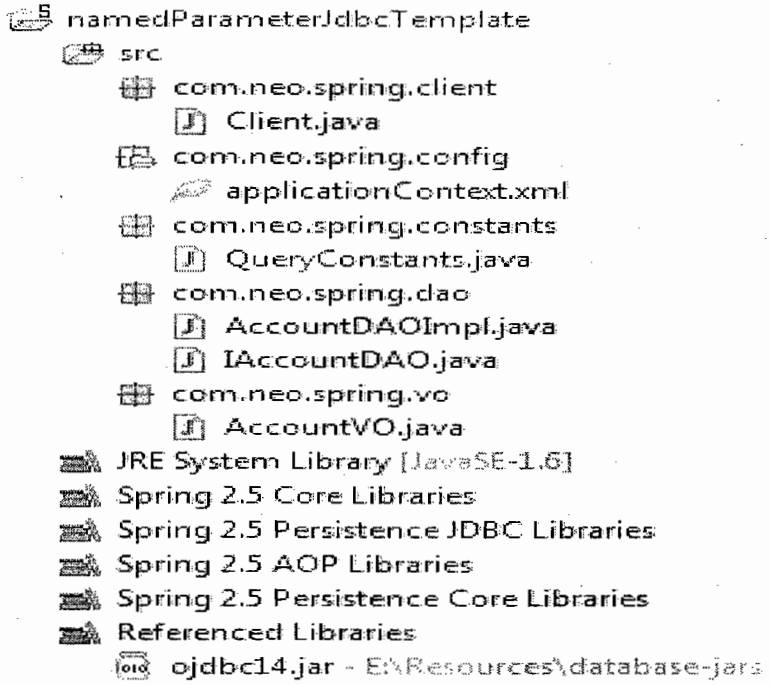
OUTPUT

```
---insert---
NO.of Rows inserted : 1
---update---
NO.of rows updated : 1
---delete---
NO.of rows deleted : 1
```

Named Parameters example with NamedParameterJdbcTemplate

In JdbcTemplate, SQL parameters are represented by a special placeholder "?" symbol and bind it by position. The problem is whenever the order of parameter is changed, you have to change the parameters bindings as well, it's error prone and cumbersome to maintain it.

To fix it, you can use "**Named Parameter**", whereas SQL parameters are defined by a starting colon follow by a name, rather than by position. In additional, the named parameters are only support in **SimpleJdbcTemplate** and **NamedParameterJdbcTemplate.**



applicationContext.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3.    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xmlns:p="http://www.springframework.org/schema/p"
5.    xsi:schemaLocation="http://www.springframework.org/schema/beans
6. http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
7.
8. <!--  Apache datasource configuration -->
9.    <bean id="apacheDatasourceRef"
10.                        class="org.apache.commons.dbcp.BasicDataSource">
11.     <property name="driverClassName"
12.                              value="oracle.jdbc.driver.OracleDriver" />
13.     <property name="url" value="jdbc:oracle:thin:@localhost:1521:XE" />
14.     <property name="username" value="system" />
15.     <property name="password" value="tiger" />
16.   </bean>
17.
18.   <bean id="NPJTRef" class="org.springframework.jdbc.core.namedparam.
19.                                          NamedParameterJdbcTemplate">
20.         <constructor-arg ref="apacheDatasourceRef" />
```

```
21.   </bean>
22.
23.   <bean id="accountDAORef" class="com.neo.spring.dao.AccountDAOImpl">
24.         <property name="namedParameterJdbcTemplate" ref="NPJTRef" />
25.   </bean>
26.
27.   </beans>
```

QueryConstants.java

```
1. package com.neo.spring.constants;
2.
3. public interface QueryConstants {
4.    public static String UPDATE_QUERY = "UPDATE ACCOUNT SET NAME= :name,
5.                                        BAL= :balance WHERE ACCNO= :accno";
6.    public static String INSERT_QUERY = "INSERT INTO ACCOUNT VALUES(:ano,
7.                                                          :name, :bal)";
8.    public static String DELETE_QUERY = "DELETE FROM ACCOUNT WHERE ACCNO=
9.                                                                 :no";
10.   }
```

AccountVO.java

```
1. package com.neo.spring.vo;
2.
3. public class AccountVO {
4.    private int accno;
5.    private String name;
6.    private double balance;
7.
8.    public AccountVO(int accno, String name, double balance) {
9.          this.accno = accno;
10.         this.name = name;
11.         this.balance = balance;
12.   }
13.    // setters & getters
14.
15.   }
```

IAccountDAO.java

```
1. package com.neo.spring.dao;
2.
3. import com.neo.spring.vo.AccountVO;
4.
5. public interface IAccountDAO {
6.    public int insert001(AccountVO accountVo);
7.    public int insert002(AccountVO accountVo);
8.    public int update(AccountVO accountVo);
9.    public int delete(int accno);
10.   }
```

AccountDAOImpl.java

```
1. package com.neo.spring.dao;
2. import java.util.HashMap;
3. import java.util.Map;
4. import
5.   org.springframework.jdbc.core.namedparam.BeanPropertySqlParameterSource;
```

```
6. import org.springframework.jdbc.core.namedparam.MapSqlParameterSource;
7. import
8. org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate;
9. import org.springframework.jdbc.core.namedparam.SqlParameterSource;
10.   import com.neo.spring.constants.QueryConstants;
11.   import com.neo.spring.vo.AccountVO;
12.   public class AccountDAOImpl implements IAccountDAO {
13.
14.         @Override
15.         public int insert001(AccountVO accountVo) {
16.               Map<String, Object> namedParams =
17.                                    new HashMap<String, Object>();
18.               namedParams.put("ano", accountVo.getAccno());
19.               namedParams.put("bal", accountVo.getBalance());
20.               namedParams.put("name", accountVo.getName());
21.
22.         return
23.         namedParameterJdbcTemplate.update(QueryConstants.INSERT_QUERY,
24.                                                              namedParams);
25.         }
26.
27.         @Override
28.         public int insert002(AccountVO accountVo) {
29.               Map<String, Object> namedParams =
30.                                  new HashMap<String, Object>();
31.               namedParams.put("ano", accountVo.getAccno());
32.               namedParams.put("bal", accountVo.getBalance());
33.               namedParams.put("name", accountVo.getName());
34.               SqlParameterSource parameterSource = new
35.                                        MapSqlParameterSource(namedParams);
36.           return
37.            namedParameterJdbcTemplate.update(QueryConstants.INSERT_QUERY,
38.                                                            parameterSource);
39.         }
40.
41.         @Override
42.         public int update(AccountVO accountVo) {
43.               SqlParameterSource paramSource = new
44.                                 BeanPropertySqlParameterSource(accountVo);
45.           return
46.            namedParameterJdbcTemplate.update(QueryConstants.UPDATE_QUERY,
47.                                                              paramSource);
48.         }
49.
50.         @Override
51.         public int delete(int accno) {
52.               SqlParameterSource parameter = new
53.                                        MapSqlParameterSource("no", accno);
54.           return
55.            namedParameterJdbcTemplate.update(QueryConstants.DELETE_QUERY,
56.                                                                parameter);
57.         }
58.
59.         private NamedParameterJdbcTemplate namedParameterJdbcTemplate;
60.
```

```
61.         public void setNamedParameterJdbcTemplate(
62.                     NamedParameterJdbcTemplate
63.                                             namedParameterJdbcTemplate) {
64.               this.namedParameterJdbcTemplate =
65.                                               namedParameterJdbcTemplate;
66.         }
67.
68.   }
```

Client.java

```
1. package com.neo.spring.client;
2. import org.springframework.context.ApplicationContext;
3. import org.springframework.context.support.ClassPathXmlApplicationContext;
4. import com.neo.spring.dao.IAccountDAO;
5. import com.neo.spring.vo.AccountVO;
6. public class Client {
7.    private static ApplicationContext context = new
8.                                          ClassPathXmlApplicationContext(
9.               "com/neo/spring/config/applicationContext.xml");
10.
11.         public static void main(String[] args) {
12.               IAccountDAO accountDAO = (IAccountDAO)
13.                                          context.getBean("accountDAORef");
14.
15.               sop("---insert001---");
16.               AccountVO accountVoA = new AccountVO(1002, "yssr", 2490.0);
17.               int ra = accountDAO.insert001(accountVoA);
18.               sop("NO.of Rows inserted : " + ra);
19.
20.               sop("---insert002---");
21.               AccountVO accountVoB = new AccountVO(1003, "sekharreddy",
22.                                                         2490.0);
23.               ra = accountDAO.insert002(accountVoB);
24.               sop("NO.of Rows inserted : " + ra);
25.
26.               sop("---update---");
27.               AccountVO accountVoC = new AccountVO(1001, "kesavareddy",
28.                                                         9490.0);
29.               ra = accountDAO.update(accountVoC);
30.               sop("NO.of rows updated : " + ra);
31.
32.               sop("---delete---");
33.               ra = accountDAO.delete(1005);
34.               sop("NO.of rows deleted : " + ra);
35.
36.         }
37.
38.         public static void sop(Object object) {
39.               System.out.println(object);
40.         }
41.
42.   }
```

Before execution Account table **After execution Account table**

ACCNO	NAME	BAL
1001	somu	8623
1005	yellareddy	4850

ACCNO	NAME	BAL
1001	kesavareddy	9490
1002	yssr	2490
1003	sekharreddy	2490

OUTPUT

```
---insert001---
NO.of Rows inserted : 1
---insert002---
NO.of Rows inserted : 1
---update---
NO.of rows updated : 1
---delete---
NO.of rows deleted : 1
```

Q) What is NamedParameterJdbcDaoSupport?

- It is very similar to **JdbcDaoSupport**.
- Instead of we writing injection logic (**NamedParameterJdbcTemplate** injection) i.e., already provided by **NamedParameterJdbcDaoSupport**.
- So every Dao class which needs **NamedParameterJdbcTemplate** injection better to extend **NamedParameterJdbcDaoSupport**.

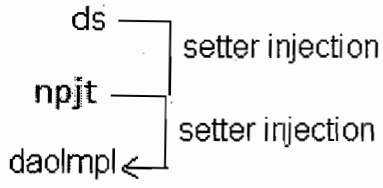
```
public class AccountDaoImpl extends NamedParameterJdbcDaoSupport {
    public void insert(Account account){
        getNamedParameterJdbcTemplate().update(query.....);
        ----------
        ----------
        // account specific dao methods
    }
}
```

Here we are calling the method **getNamedParameterJdbcTemplate()** to get **NamedParameterJdbcTemplate** object. This method is available in **NamedParameterJdbcDaoSupport** class.

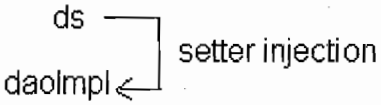
Q) What is the flexibility given by NamedParameterJdbcDaoSupport to configure Dao classes in spring configuration file?

Ans: Instead of injecting **NamedParameterJdbcTemplate** object into the Dao class, we can inject **DataSource** in the constructor approach.

Without NamedParameterJdbcDaoSupport



With NamedParameterJdbcDaoSupport



Named Parameters example with SimpleJdbcTemplate

In JdbcTemplate, SQL parameters are represented by a special placeholder "?" symbol and bind it by position. The problem is whenever the order of parameter is changed, you have to change the parameters bindings as well, it's error prone and cumbersome to maintain it.

To fix it, you can use "**Named Parameter**", whereas SQL parameters are defined by a starting colon follow by a name, rather than by position. In additional, the named parameters are only support in **SimpleJdbcTemplate** and **NamedParameterJdbcTemplate**.

applicationContext.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3.     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xmlns:p="http://www.springframework.org/schema/p"
5.     xsi:schemaLocation="http://www.springframework.org/schema/beans
6. http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
7.
8. <!--  Apache datasource configuration -->
9.     <bean id="apacheDatasourceRef"
10.                          class="org.apache.commons.dbcp.BasicDataSource">
11.      <property name="driverClassName"
12.                                value="oracle.jdbc.driver.OracleDriver" />
13.      <property name="url" value="jdbc:oracle:thin:@localhost:1521:XE" />
14.      <property name="username" value="system" />
15.      <property name="password" value="tiger" />
16.    </bean>
17.
18.    <bean id="simpleJTRef"
19.        class="org.springframework.jdbc.core.simple.SimpleJdbcTemplate">
20.          <constructor-arg ref="apacheDatasourceRef"></constructor-arg>
21.    </bean>
22.
23.    <bean id="accountDAORef" class="com.neo.spring.dao.AccountDAOImpl">
24.          <property name="simpleJdbcTemplate" ref="simpleJTRef"></property>
25.    </bean>
26.
27.    </beans>
```

QueryConstants.java

```
1. package com.neo.spring.constants;
2.
3. public interface QueryConstants {
4.     public static String UPDATE_QUERY = "UPDATE ACCOUNT SET NAME=
5.                           :name, BAL= :balance WHERE ACCNO= :accno";
6.     public static String INSERT_QUERY = "INSERT INTO ACCOUNT
7.                                          VALUES(:ano, :name, :bal)";
8.     public static String DELETE_QUERY = "DELETE FROM ACCOUNT WHERE
9.                                                    ACCNO= :no";
10.    }
```

AccountVO.java

```
1. package com.neo.spring.vo;
```

```
2.
3. public class AccountVO {
4.     private int accno;
5.     private String name;
6.     private double balance;
7.
8.     public AccountVO(int accno, String name, double balance) {
9.           this.accno = accno;
10.          this.name = name;
11.          this.balance = balance;
12.    }
13.
14.    //setters & getters
15.
16.    }
```

IAccountDAO.java

```
1. package com.neo.spring.dao;
2.
3. import com.neo.spring.vo.AccountVO;
4.
5. public interface IAccountDAO {
6.     public int insert001(AccountVO accountVo);
7.     public int insert002(AccountVO accountVo);
8.     public int update(AccountVO accountVo);
9.     public int delete(int accno);
10.    }
```

AccountDAOImpl.java

```
1. package com.neo.spring.dao;
2.
3. import java.util.HashMap;
4. import java.util.Map;
5.
6. import
7. org.springframework.jdbc.core.namedparam.BeanPropertySqlParameterSource
8. ;
9. import org.springframework.jdbc.core.namedparam.MapSqlParameterSource;
10.   import org.springframework.jdbc.core.namedparam.SqlParameterSource;
11.   import org.springframework.jdbc.core.simple.SimpleJdbcTemplate;
12.
13.   import com.neo.spring.constants.QueryConstants;
14.   import com.neo.spring.vo.AccountVO;
15.
16.   public class AccountDAOImpl implements IAccountDAO {
17.
18.         @Override
19.         public int insert001(AccountVO accountVo) {
20.               Map<String, Object> namedParams = new HashMap<String,
21.                                                          Object>();
22.               namedParams.put("ano", accountVo.getAccno());
23.               namedParams.put("bal", accountVo.getBalance());
24.               namedParams.put("name", accountVo.getName());
25.
26.               return
```

```
27.               simpleJdbcTemplate.update(QueryConstants.INSERT_QUERY,
28.                                                         namedParams);
29.         }
30.
31.         @Override
32.         public int insert002(AccountVO accountVo) {
33.               Map<String, Object> namedParams = new HashMap<String,
34.                                                         Object>();
35.               namedParams.put("ano", accountVo.getAccno());
36.               namedParams.put("bal", accountVo.getBalance());
37.               namedParams.put("name", accountVo.getName());
38.               SqlParameterSource parameterSource = new
39.                                    MapSqlParameterSource(namedParams);
40.
41.               return
42.                     simpleJdbcTemplate.update(QueryConstants.INSERT_QUERY,
43.                                                         parameterSource);
44.         }
45.
46.         @Override
47.         public int update(AccountVO accountVo) {
48.               SqlParameterSource paramSource = new
49.                               BeanPropertySqlParameterSource(accountVo);
50.               return
51.                     simpleJdbcTemplate.update(QueryConstants.UPDATE_QUERY,
52.                                                         paramSource);
53.         }
54.
55.         @Override
56.         public int delete(int accno) {
57.               SqlParameterSource parameter = new
58.                                    MapSqlParameterSource("no", accno);
59.               return
60.                     simpleJdbcTemplate.update(QueryConstants.DELETE_QUERY,
61.                                                         parameter);
62.         }
63.
64.         private SimpleJdbcTemplate simpleJdbcTemplate;
65.
66.         public void setSimpleJdbcTemplate(SimpleJdbcTemplate
67.                                                   simpleJdbcTemplate) {
68.               this.simpleJdbcTemplate = simpleJdbcTemplate;
69.         }
70.
71.   }
```

Client.java

```
1. package com.neo.spring.client;
2.
3. import org.springframework.context.ApplicationContext;
4. import
5.     org.springframework.context.support.ClassPathXmlApplicationContext;
6. import com.neo.spring.dao.IAccountDAO;
7. import com.neo.spring.vo.AccountVO;
```

```
8.
9. public class Client {
10.         private static ApplicationContext context = new
11.                                             ClassPathXmlApplicationContext(
12.                     "com/neo/spring/config/applicationContext.xml");
13.
14.         public static void main(String[] args) {
15.               IAccountDAO accountDAO = (IAccountDAO)
16.                                         context.getBean("accountDAORef");
17.
18.               sop("---insert001---");
19.               AccountVO accountVoA = new AccountVO(1002, "yssr", 2490.0);
20.               int ra = accountDAO.insert001(accountVoA);
21.               sop("NO.of Rows inserted : " + ra);
22.
23.               sop("---insert002---");
24.               AccountVO accountVoB = new AccountVO(1003, "sekharreddy",
25.                                                                   2490.0);
26.               ra = accountDAO.insert002(accountVoB);
27.               sop("NO.of Rows inserted : " + ra);
28.
29.
30.               sop("---update---");
31.               AccountVO accountVoC = new AccountVO(1001, "kesavareddy",
32.                                                                   9490.0);
33.               ra = accountDAO.update(accountVoC);
34.               sop("NO.of rows updated : " + ra);
35.
36.               sop("---delete---");
37.               ra = accountDAO.delete(1005);
38.               sop("NO.of rows deleted : " + ra);
39.
40.         }
41.
42.         public static void sop(Object object) {
43.               System.out.println(object);
44.         }
45.   }
```

Before execution Account table

ACCNO	NAME	BAL
1001	somu	8623
1005	yellareddy	4856

After execution Account table

ACCNO	NAME	BAL
1001	kesavareddy	9490
1002	yssr	2490
1003	sekharreddy	2490

OUTPUT

```
---insert001---
NO.of Rows inserted : 1
---insert002---
NO.of Rows inserted : 1
---update---
NO.of rows updated : 1
---delete---
NO.of rows deleted : 1
```

queryForInt()

public int queryForInt(String sql):-

This method executes the query and gives the result as an int value. The result of the query is passed t this method is expected to be return a single row and single column. If the query does not returning exactly one row, it throws "**org.springframework.dao.IncorrectResultSizeDataAccessException**"

And if the query does not returning exactly one column it throws

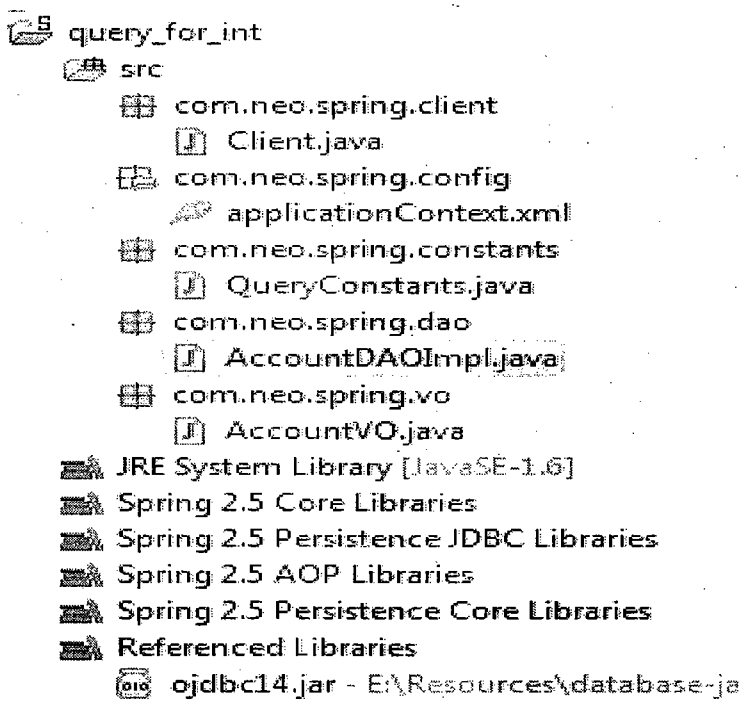
"**org.springframework.jdbc.IncorrectResultSetColumnCountException**"

public int queryForInt(String sql, Object[] args):-

This method is similar to **queryForInt(String sql)** but this method have second argument that is used for give the values to the place holders of the query.

public int queryForInt(String sql, Object[] args, int[] argTypes):-

This method is also same as above methods. The third argument will take the SQl types of the place holders.



applicationContext.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3.     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xmlns:p="http://www.springframework.org/schema/p"
5.     xsi:schemaLocation="http://www.springframework.org/schema/beans
6. http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
7.
8. <!--  Apache datasource configuration -->
9.     <bean id="apacheDatasourceRef"
10.                         class="org.apache.commons.dbcp.BasicDataSource">
```

```
11.     <property name="driverClassName"
12.                                 value="oracle.jdbc.driver.OracleDriver" />
13.     <property name="url" value="jdbc:oracle:thin:@localhost:1521:XE" />
14.     <property name="username" value="system" />
15.     <property name="password" value="tiger" />
16.   </bean>
17.
18.   <bean id="accountDAORef" class="com.neo.spring.dao.AccountDAOImpl">
19.         <property name="dataSource" ref=" apacheDatasourceRef " />
20.   </bean>
21.
22.   </beans>
```

QueryConstants.java

```
1. package com.neo.spring.constants;
2.
3. public interface QueryConstants {
4.     public static String SELECT_QUERY_001 = "SELECT COUNT(*) FROM ACCOUNT";
5.     public static String SELECT_QUERY_002 = "SELECT COUNT(*) FROM ACCOUNT
6.                                                 WHERE BAL>100";
7.     public static String SELECT_QUERY_003 = "SELECT COUNT(*) FROM ACCOUNT
8.                                                   WHERE BAL>?";
9.     public static String SELECT_QUERY_004 = "SELECT COUNT(*) FROM ACCOUNT
10.                                            WHERE BAL>? AND NAME LIKE ?";
11.    public static String SELECT_QUERY_005 = "SELECT ACCNO FROM ACCOUNT
12.                                                 WHERE ACCNO = ?";
13.    public static String SELECT_QUERY_006 = "SELECT ACCNO FROM ACCOUNT";
14.    public static String SELECT_QUERY_007 = "SELECT NAME,BAL FROM ACCOUNT";
15.    public static String SELECT_QUERY_008 = "SELECT NAME FROM ACCOUNT WHERE
16.                                                   ACCNO=1005";
17.    }
```

AccountVO.java

```
1. package com.neo.spring.vo;
2.
3. public class AccountVO {
4.     private int accno;
5.     private String name;
6.     private double balance;
7.
8.        public AccountVO(){
9.        }
10.
11.       public AccountVO(int accno, String name, double balance) {
12.               this.accno = accno;
13.               this.name = name;
14.               this.balance = balance;
15.          }
16.          //setters & getters
17.
18.   }
```

AccountDAOImpl.java

```
1. package com.neo.spring.dao;
2.
```

```
3. import java.sql.Types;
4.
5. import org.springframework.jdbc.core.support.JdbcDaoSupport;
6.
7. import com.neo.spring.constants.QueryConstants;
8.
9. public class AccountDAOImpl extends JdbcDaoSupport{
10.
11.         public int usingQueryForInt001(){
12.               return getJdbcTemplate().queryForInt(
13.                                          QueryConstants.SELECT_QUERY_001);
14.         }
15.
16.         public int usingQueryForInt002(){
17.               return getJdbcTemplate().queryForInt(
18.                                          QueryConstants.SELECT_QUERY_002);
19.         }
20.
21.         public int usingQueryForInt003(){
22.               return getJdbcTemplate().queryForInt(
23.                      QueryConstants.SELECT_QUERY_003, new Object[]{150});
24.         }
25.
26.         public int usingQueryForInt004(){
27.               return getJdbcTemplate().queryForInt(
28.                     QueryConstants.SELECT_QUERY_004, new Object[]{150,
29.                     "%sekhar%"}, new int[]{Types.DOUBLE, Types.VARCHAR});
30.         }
31.
32.         public int usingQueryForInt005(){
33.               return getJdbcTemplate().queryForInt(
34.                      QueryConstants.SELECT_QUERY_005, new Object[]{1005},
35.                                                 new int[]{Types.INTEGER});
36.         }
37.
38.         public int usingQueryForInt006(){
39.               return getJdbcTemplate().queryForInt(
40.                                          QueryConstants.SELECT_QUERY_006);
41.         }
42.
43.         public int usingQueryForInt007(){
44.               return getJdbcTemplate().queryForInt(
45.                                          QueryConstants.SELECT_QUERY_007);
46.         }
47.
48.         public void usingQueryForInt008(){
49.
50.           getJdbcTemplate().queryForInt(QueryConstants.SELECT_QUERY_008);
51.         }
52.
53.   }
```

Client.java

```
1. package com.neo.spring.client;
2.
```

```
3. import org.springframework.context.ApplicationContext;
4. import org.springframework.context.support.ClassPathXmlApplicationContext;
5.
6. import com.neo.spring.dao.AccountDAOImpl;
7.
8. public class Client {
9.    private static ApplicationContext context = new
10.                  ClassPathXmlApplicationContext(
11.                     "com/neo/spring/config/applicationContext.xml");
12.
13.       public static void main(String[] args) {
14.             AccountDAOImpl impl= (AccountDAOImpl)
15.                                    context.getBean("accountDAORef");
16.
17.             sop("---usingQueryForInt001---");
18.             int rowCount = impl.usingQueryForInt001();
19.             sop("No.of rows :"+rowCount);
20.
21.             sop("---usingQueryForInt002---");
22.             rowCount = impl.usingQueryForInt002();
23.             sop("No.of rows :"+rowCount);
24.
25.             sop("---usingQueryForInt003---");
26.             rowCount = impl.usingQueryForInt003();
27.             sop("No.of rows :"+rowCount);
28.
29.             sop("---usingQueryForInt004---");
30.             rowCount = impl.usingQueryForInt004();
31.             sop("No.of rows :"+rowCount);
32.
33.             sop("---usingQueryForInt005---");
34.             rowCount = impl.usingQueryForInt005();
35.             sop("ACCNO :"+rowCount);
36.
37.   //        sop("---usingQueryForInt006---");
38.   //        int rowCount = impl.usingQueryForInt006();
39.   //        sop("No.of rows :"+rowCount);
40.
41.   //        sop("---usingQueryForInt007---");
42.   //        int rowCount = impl.usingQueryForInt007();
43.   //        sop("No.of rows :"+rowCount);
44.
45.   //        sop("---usingQueryForInt008---");
46.   //        impl.usingQueryForInt008();
47.
48.
49.       }
50.
51.       public static void sop(Object object) {
52.             System.out.println(object);
53.       }
54.
55.  }
```

Before execution Account table

ACCNO	NAME	BAL
1004	somu	9800
1005	sekharreddy	2490
1003	somasekhar	7890

OUTPUT

```
---usingQueryForInt001---
No.of rows :3
---usingQueryForInt002---
No.of rows :3
---usingQueryForInt003---
No.of rows :3
---usingQueryForInt004---
No.of rows :2
---usingQueryForInt005---
ACCNO :1005
```

NOTE:

Observe that in the methods

dao.usingQueryForInt006() throws "org.springframework.dao.IncorrectResultSizeDataAccessException" because the result will be two rows.

dao.usingQueryForInt007() throws

"org.springframework.jdbc.IncorrectResultSetColumnCountException" because the result wil be two columns.

dao.usingQueryForInt008() throws

org.springframework.jdbc.UncategorizedSQLException : Fail to convert to internal representation

queryForObject()

public Object queryForObject(String sql, Class requiredType):-

It execute a query for a result object. The query is expected to be a single row and single column. Second argument will takes the class constant it specifies that the type of result. The returned result will be directly mapped to the corresponding object type. If the query does not returning exactly one row, it throws "**org.springframework.dao.IncorrectResultSizeDataAccessException**". And if the query does not returning exactly one column it throws "**org.springframework.jdbc.IncorrectResultSetColumnCountException**"

public Object queryForObject(String sql, Object[] args, Class requiredType):-

It is same as above method and it also have Object[] which takes the values for place holders.

public Object queryForObject(String sql, Object[] args, int[] argTypes, Class requiredType):-

This method is also same as above methods. The third argument will take the SQl type.

public <T> T queryForObject(String sql, RowMapper<T> rowMapper):-

This method is used to execute a query, and mapping a single result row to a Java object via a RowMapper. RowMapper is an interface and it has one method **public** Object mapRow(ResultSet rs, **int** rowNum).

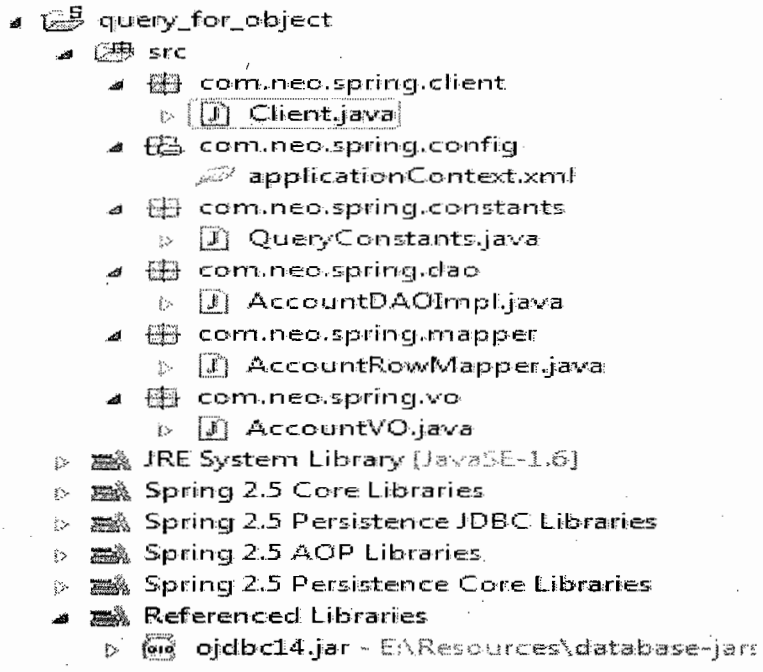
If we want to use RowMapper, We have to implement this RowMapper and override this mapRow method. RowMapper will used to map the resultset to our domain object or pojo or DTO.
if the query does not return exactly one row it throws
"**org.springframework.dao.IncorrectResultSizeDataAccessException**"

<T> T queryForObject(String sql, Object[] args, RowMapper<T> rowMapper):-

This method is same of above method and also it is used for giving the values to place holders.

<T> T queryForObject(String sql, Object[] args, int[] argTypes, RowMapper<T> rowMapper):-

This method is same of above method and specifying their types.



applicationContext.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3.     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xmlns:p="http://www.springframework.org/schema/p"
5.     xsi:schemaLocation="http://www.springframework.org/schema/beans
6. http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
7.
8. <!--  Apache datasource configuration -->
9.     <bean id="apacheDatasourceRef"
10.                          class="org.apache.commons.dbcp.BasicDataSource">
11.      <property name="driverClassName"
12.                                  value="oracle.jdbc.driver.OracleDriver" />
13.      <property name="url" value="jdbc:oracle:thin:@localhost:1521:XE" />
14.      <property name="username" value="system" />
15.      <property name="password" value="tiger" />
16.    </bean>
17.
18.    <bean id="accountDAORef" class="com.neo.spring.dao.AccountDAOImpl">
19.          <property name="dataSource" ref=" apacheDatasourceRef " />
20.    </bean>
21.
22.    </beans>
```

QueryConstants.java

```
1. package com.neo.spring.constants;
2.
3. public interface QueryConstants {
4.     public static String SELECT_QUERY_001 = "SELECT COUNT(*) FROM ACCOUNT";
5.     public static String SELECT_QUERY_002 = "SELECT NAME FROM ACCOUNT WHERE
6.                                                          ACCNO = 1005";
7.     public static String SELECT_QUERY_003 = "SELECT NAME FROM ACCOUNT WHERE
8.                                                            ACCNO = ?";
```

```
9.    public static String SELECT_QUERY_004 = "SELECT NAME FROM ACCOUNT WHERE
10.                                                    ACCNO = ? AND BAL>?";
11.   public static String SELECT_QUERY_005 = "SELECT ACCNO, NAME, BAL
12.                                           FROM ACCOUNT WHERE ACCNO = 1005";
13.   public static String SELECT_QUERY_006 = "SELECT * FROM ACCOUNT WHERE
14.                                                        ACCNO=1005";
15.   public static String SELECT_QUERY_007 = "SELECT * FROM ACCOUNT WHERE
16.                                                          ACCNO=?";
17.   public static String SELECT_QUERY_008 = "SELECT ACCNO, NAME, BAL FROM
18.                                                  ACCOUNT WHERE ACCNO=?";
19.   public static String SELECT_QUERY_009 = "SELECT ACCNO, NAME, BAL FROM
20.                                              ACCOUNT WHERE ACCNO=1005";
21.   public static String SELECT_QUERY_010 = "SELECT ACCNO, NAME, BAL as
22.                                    balance FROM ACCOUNT WHERE ACCNO=1005";
23.   public static String SELECT_QUERY_011 = "SELECT NAME FROM ACCOUNT";
24.   public static String SELECT_QUERY_012 = "SELECT NAME, BAL FROM ACCOUNT
25.                                                     WHERE ACCNO=1005";
26.   public static String SELECT_QUERY_013 = "SELECT ACCNO, NAME, BAL as
27.                                                   balance FROM ACCOUNT";
28.   }
```

AccountVO.java

```
1. package com.neo.spring.vo;
2.
3. public class AccountVO {
4.    private int accno;
5.    private String name;
6.    private double balance;
7.
8.       public AccountVO(){
9.       }
10.
11.      public AccountVO(int accno, String name, double balance) {
12.               this.accno = accno;
13.               this.name = name;
14.               this.balance = balance;
15.         }
16.         //setters & getters
17.
18.   }
```

AccountDAOImpl.java

```
1. package com.neo.spring.dao;
2.
3. import java.sql.ResultSet;
4. import java.sql.SQLException;
5. import java.sql.Types;
6.
7. import org.springframework.jdbc.core.BeanPropertyRowMapper;
8. import org.springframework.jdbc.core.RowMapper;
9. import org.springframework.jdbc.core.support.JdbcDaoSupport;
10.
11.   import com.neo.spring.constants.QueryConstants;
12.   import com.neo.spring.mapper.AccountRowMapper;
13.   import com.neo.spring.vo.AccountVO;
```

```
14.
15.  public class AccountDAOImpl extends JdbcDaoSupport{
16.
17.        public int usingQueryForObject001(){
18.              return (Integer)getJdbcTemplate().queryForObject(
19.                        QueryConstants.SELECT_QUERY_001, Integer.class);
20.
21.        }
22.
23.        public String usingQueryForObject002(){
24.              return (String)getJdbcTemplate().queryForObject(
25.                        QueryConstants.SELECT_QUERY_002, String.class);
26.        }
27.
28.        public String usingQueryForObject003(){
29.              return (String)getJdbcTemplate().queryForObject(
30.                       QueryConstants.SELECT_QUERY_003, new Object[]{1005},
31.                                                              String.class);
32.        }
33.
34.        public String usingQueryForObject004(){
35.              return (String)getJdbcTemplate().queryForObject(
36.                        QueryConstants.SELECT_QUERY_004, new Object[]{1005,
37.              200},new int[]{Types.INTEGER, Types.DOUBLE} ,String.class);
38.        }
39.
40.        public AccountVO usingQueryForObject005(){
41.              return (AccountVO)getJdbcTemplate().queryForObject(
42.                QueryConstants.SELECT_QUERY_005, new AccountRowMapper());
43.        }
44.
45.        public AccountVO usingQueryForObject006(){
46.              return (AccountVO)getJdbcTemplate().queryForObject(
47.                       QueryConstants.SELECT_QUERY_006,
48.              new RowMapper() {
49.
50.              @Override
51.              public Object mapRow(ResultSet rs, int rowCount)
52.                                                  throws SQLException {
53.               System.out.println("RowCount : "+rowCount);
54.                    AccountVO accountVO = new AccountVO();
55.                    accountVO.setAccno(rs.getInt(1));
56.                    accountVO.setName(rs.getString(2));
57.                    accountVO.setBalance(rs.getDouble(3));
58.
59.                    return accountVO;
60.                    }
61.              });
62.        }
63.
64.        public AccountVO usingQueryForObject007(){
65.              return (AccountVO)getJdbcTemplate().queryForObject(
66.                       QueryConstants.SELECT_QUERY_007, new Object[]{1005}
67.                     , new AccountRowMapper());
68.        }
```

```
69.
70.         public AccountVO usingQueryForObject008(){
71.               return (AccountVO)getJdbcTemplate().queryForObject(
72.                     QueryConstants.SELECT_QUERY_008, new Object[]{1005},
73.                      new int[]{Types.INTEGER} , new AccountRowMapper());
74.         }
75.
76.         public AccountVO usingQueryForObject009(){
77.               return (AccountVO)getJdbcTemplate().queryForObject(
78.                     QueryConstants.SELECT_QUERY_009, new
79.                     BeanPropertyRowMapper(AccountVO.class));
80.         }
81.
82.         public AccountVO usingQueryForObject010(){
83.               return (AccountVO)getJdbcTemplate().queryForObject(
84.                      QueryConstants.SELECT_QUERY_010, new
85.                       BeanPropertyRowMapper(AccountVO.class));
86.         }
87.
88.         public void usingQueryForObject011(){
89.               getJdbcTemplate().queryForObject(
90.                     QueryConstants.SELECT_QUERY_011, String.class);
91.         }
92.
93.         public void usingQueryForObject012(){
94.               getJdbcTemplate().queryForObject(
95.                     QueryConstants.SELECT_QUERY_012, String.class);
96.         }
97.
98.         public void usingQueryForObject013(){
99.               getJdbcTemplate().queryForObject(
100.                    c              QueryConstants.SELECT_QUERY_013, new
101.                                BeanPropertyRowMapper(AccountVO.class));
102.        }
103. }
```

AccountRowMapper.java

```
1. package com.neo.spring.mapper;
2.
3. import java.sql.ResultSet;
4. import java.sql.SQLException;
5.
6. import org.springframework.jdbc.core.RowMapper;
7.
8. import com.neo.spring.vo.AccountVO;
9.
10.   public class AccountRowMapper implements RowMapper{
11.    @Override
12.   public Object mapRow(ResultSet rs, int rowCount) throws SQLException {
13.         AccountVO accountVO = new AccountVO();
14.         accountVO.setAccno(rs.getInt(1));
15.         accountVO.setName(rs.getString(2));
16.         accountVO.setBalance(rs.getDouble(3));
17.
18.         return accountVO;
```

```
19.   }
20.   }
```

Client.java

```
1. package com.neo.spring.client;
2.
3. import org.springframework.context.ApplicationContext;
4. import org.springframework.context.support.ClassPathXmlApplicationContext;
5.
6. import com.neo.spring.dao.AccountDAOImpl;
7. import com.neo.spring.vo.AccountVO;
8.
9. public class Client {
10.         private static ApplicationContext context = new
11.              ClassPathXmlApplicationContext(
12.                    "com/neo/spring/config/applicationContext.xml");
13.
14.         public static void main(String[] args) {
15.               AccountDAOImpl impl= (AccountDAOImpl)
16.                                    context.getBean("accountDAORef");
17.
18.               sop("\n---usingQueryForObject001---");
19.               int rowCount = impl.usingQueryForObject001();
20.               sop("No.of rows :"+rowCount);
21.
22.               sop("\n---usingQueryForObject002---");
23.               String name = impl.usingQueryForObject002();
24.               sop("Name :"+name);
25.
26.               sop("\n---usingQueryForObject003---");
27.               name = impl.usingQueryForObject003();
28.               sop("Name :"+name);
29.
30.               sop("\n---usingQueryForObject004---");
31.               name = impl.usingQueryForObject004();
32.               sop("Name :"+name);
33.
34.               sop("\n---usingQueryForObject005---");
35.               AccountVO accountVO = impl.usingQueryForObject005();
36.               sop("Account details ....");
37.               sop("ACCNO : "+accountVO.getAccno());
38.               sop("NAME : "+accountVO.getName());
39.               sop("BALANCE : "+accountVO.getBalance());
40.
41.               sop("\n---usingQueryForObject006---");
42.               accountVO = impl.usingQueryForObject006();
43.               sop("Account details ....");
44.               sop("ACCNO : "+accountVO.getAccno());
45.               sop("NAME : "+accountVO.getName());
46.               sop("BALANCE : "+accountVO.getBalance());
47.
48.               sop("\n---usingQueryForObject007---");
49.               accountVO = impl.usingQueryForObject007();
50.               sop("Account details ....");
51.               sop("ACCNO : "+accountVO.getAccno());
```

```
52.                 sop("NAME : "+accountVO.getName());
53.                 sop("BALANCE : "+accountVO.getBalance());
54.
55.                 sop("\n---usingQueryForObject008---");
56.                 accountVO = impl.usingQueryForObject008();
57.                 sop("Account details ....");
58.                 sop("ACCNO : "+accountVO.getAccno());
59.                 sop("NAME : "+accountVO.getName());
60.                 sop("BALANCE : "+accountVO.getBalance());
61.
62.                 sop("\n---usingQueryForObject009---");
63.                 accountVO = impl.usingQueryForObject009();
64.                 sop("Account details ....");
65.                 sop("ACCNO : "+accountVO.getAccno());
66.                 sop("NAME : "+accountVO.getName());
67.                 sop("BALANCE : "+accountVO.getBalance());
68.
69.                 sop("\n---usingQueryForObject010---");
70.                 accountVO = impl.usingQueryForObject010();
71.                 sop("Account details ....");
72.                 sop("ACCNO : "+accountVO.getAccno());
73.                 sop("NAME : "+accountVO.getName());
74.                 sop("BALANCE : "+accountVO.getBalance());
75.
76.                 sop("\n---usingQueryForObject0011---");
77.                 impl.usingQueryForObject011();
78.     /*          IncorrectResultSizeDataAccessException */
79.
80.                 sop("\n---usingQueryForObject012---");
81.                 impl.usingQueryForObject012();
82.     /*          IncorrectResultSetColumnCountException  */
83.
84.                 sop("\n---usingQueryForObject013---");
85.                 impl.usingQueryForObject013();
86.     /*          IncorrectResultSizeDataAccessException*/
87.         }
88.
89.         public static void sop(Object object) {
90.                 System.out.println(object);
91.         }
92.
93.     }
```

Before execution Account table

ACCNO	NAME	BAL
1004	somu	9800
1005	sekharreddy	2490
1003	somasekhar	7890

queryForMap()

Map queryForMap(String sql):-

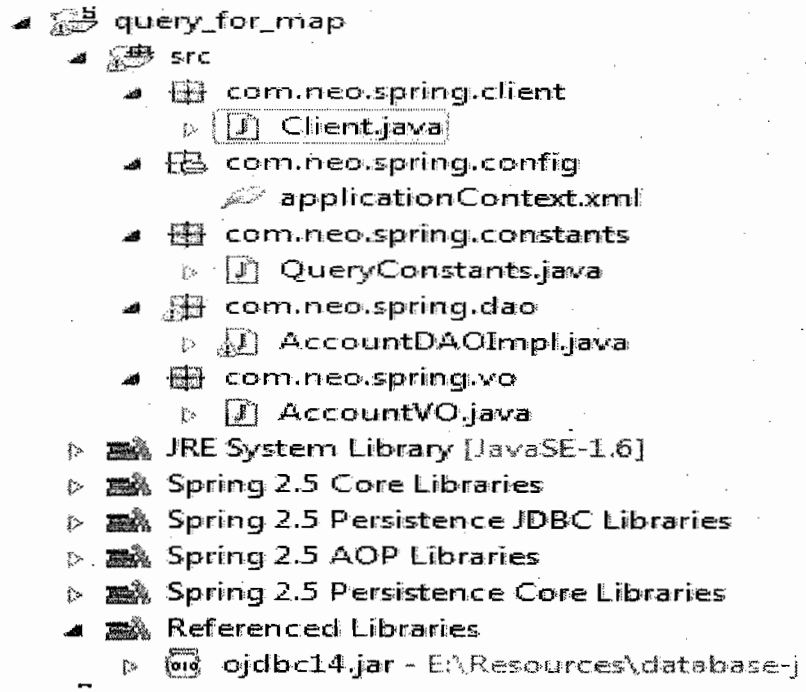
The queryForMap method, only expects a single row to be returned. The return value for this method will be a map of column names to column values for the single result row. Column name as the key and column value as the value. If we try to return multiple rows it throws **org.springframework.dao.IncorrectResultSizeDataAccessException.** The query returns exactly one row.

Map queryForMap(String sql, Object[] args):-

This method is similar to above method and in second argument we will pass the values for place holders. It also returns exactly single row and If we try to return multiple rows it throws **org.springframework.dao.IncorrectResultSizeDataAccessException**.

Map queryForMap(String sql, Object[] args, int[] argTypes):-

This method also similar to above methods and it has another argument. This argument takes the type of the place holder values. It also returns exactly single row. If we try to return multiple rows it throws **org.springframework.dao.IncorrectResultSizeDataAccessException**.



applicationContext.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3.     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xmlns:p="http://www.springframework.org/schema/p"
5.     xsi:schemaLocation="http://www.springframework.org/schema/beans
6. http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
7.
```

```
8. <!--  Apache datasource configuration -->
9.     <bean id="apacheDatasourceRef"
10.                          class="org.apache.commons.dbcp.BasicDataSource">
11.      <property name="driverClassName"
12.                                  value="oracle.jdbc.driver.OracleDriver" />
13.      <property name="url" value="jdbc:oracle:thin:@localhost:1521:XE" />
14.      <property name="username" value="system" />
15.      <property name="password" value="tiger" />
16.    </bean>
17.
18.    <bean id="accountDAORef" class="com.neo.spring.dao.AccountDAOImpl">
19.          <property name="dataSource" ref=" apacheDatasourceRef " />
20.    </bean>
21.
22.    </beans>
```

QueryConstants.java

```
1. package com.neo.spring.constants;
2. public interface QueryConstants {
3.     public static String SELECT_QUERY_001 = "SELECT NAME FROM ACCOUNT WHERE
4.                                                          ACCNO=1005" ;
5.     public static String SELECT_QUERY_002 = "SELECT ACCNO, NAME, BAL FROM
6.                                                  ACCOUNT WHERE ACCNO=1005" ;
7.     public static String SELECT_QUERY_003 = "SELECT ACCNO, NAME, BAL FROM
8.                                                    ACCOUNT WHERE ACCNO=?" ;
9. public static String SELECT_QUERY_004 = "SELECT * FROM ACCOUNT WHERE
10.                                                          ACCNO=?" ;
11.    public static String SELECT_QUERY_005 = "SELECT ACCNO, NAME, BAL
12.                                                       FROM ACCOUNT" ;
13.    }
```

AccountVO.java

```
1. package com.neo.spring.vo;
2. public class AccountVO {
3.     private int accno;
4.     private String name;
5.     private double balance;
6.
7.        public AccountVO(){
8.        }
9.
10.       public AccountVO(int accno, String name, double balance) {
11.                this.accno = accno;
12.                this.name = name;
13.                this.balance = balance;
14.          }
15.          //setters & getters
16.    }
```

AccountDAOImpl.java

```
1. package com.neo.spring.dao;
2. import java.sql.Types;
3. import java.util.Map;
4. import java.util.Set;
5. import org.springframework.jdbc.core.support.JdbcDaoSupport;
```

```
6. import com.neo.spring.constants.QueryConstants;
7. public class AccountDAOImpl extends JdbcDaoSupport{
8.
9.     public Map<String, Object> usingQueryForMap001(){
10.              return getJdbcTemplate().queryForMap(
11.                                        QueryConstants.SELECT_QUERY_001);
12.         }
13.
14.         public Map<String, Object> usingQueryForMap002(){
15.              return getJdbcTemplate().queryForMap(
16.                                       QueryConstants.SELECT_QUERY_002);
17.         }
18.
19.         public Map<String, Object> usingQueryForMap003(){
20.              return getJdbcTemplate().queryForMap(
21.                    QueryConstants.SELECT_QUERY_003, new Object[]{1005});
22.         }
23.
24.         public Map<String, Object> usingQueryForMap004(){
25.              return getJdbcTemplate().queryForMap(
26.                     QueryConstants.SELECT_QUERY_004, new Object[]{1005},
27.                                              new int[]{Types.INTEGER});
28.         }
29.
30.         public Map<String, Object> usingQueryForMap005(){
31.              return getJdbcTemplate().queryForMap(
32.                                        QueryConstants.SELECT_QUERY_005);
33.         }
34.   }
```

Client.java

```
1. package com.neo.spring.client;
2. import java.util.Iterator;
3. import java.util.Map;
4. import java.util.Set;
5. import java.util.Map.Entry;
6. import org.springframework.context.ApplicationContext;
7. import org.springframework.context.support.ClassPathXmlApplicationContext;
8. import com.neo.spring.dao.AccountDAOImpl;
9. public class Client {
10.         private static ApplicationContext context = new
11.                     ClassPathXmlApplicationContext(
12.                     "com/neo/spring/config/applicationContext.xml");
13.
14.         public static void main(String[] args) {
15.              AccountDAOImpl impl= (AccountDAOImpl)
16.                                  context.getBean("accountDAORef");
17.
18.              sop("\n---usingQueryForMap001---");
19.              Map<String, Object> map = impl.usingQueryForMap001();
20.              sop(map);
21.              Set<String> keys = map.keySet();
22.              Iterator<String> itr = keys.iterator();
23.              while(itr.hasNext()){
24.                   String key = itr.next();
```

```
25.                     sop(key+"==>"+map.get(key));
26.               }
27.
28.
29.               sop("\n---usingQueryForMap002---");
30.               map= impl.usingQueryForMap002();
31.               for(String key : map.keySet())
32.                     sop(key+"==>"+map.get(key));
33.
34.               sop("\n---usingQueryForMap003---");
35.               map=impl.usingQueryForMap003();
36.               Set<Entry<String, Object>> entries = map.entrySet();
37.               for(Entry<String, Object> entry : entries)
38.                     sop(entry.getKey()+"==>"+entry.getValue());
39.
40.               sop("\n---usingQueryForMap004---");
41.               map = impl.usingQueryForMap004();
42.               for(String key : map.keySet()){
43.                     sop(key+"==>"+map.get(key));
44.               }
45.
46.   //          sop("\n---usingQueryForMap005---");
47.   //          impl.usingQueryForMap005();
48.   /*          IncorrectResultSizeDataAccessException*/
49.         }
50.
51.         public static void sop(Object object) {
52.               System.out.println(object);
53.         }
54.   }
```

Before execution Account table

ACCNO	NAME	BAL
1004	somu	9800
1005	sekharreddy	2490
1003	somasekhar	7890

Output:

```
---usingQueryForMap001---
{NAME=sekharreddy}
NAME==>sekharreddy

---usingQueryForMap002---
ACCNO==>1005
NAME==>sekharreddy
BAL==>2490

---usingQueryForMap003---
ACCNO==>1005
NAME==>sekharreddy
BAL==>2490

---usingQueryForMap004---
ACCNO==>1005
NAME==>sekharreddy
BAL==>2490
```

queryForList()

public List queryForList(String sql):-

This method is used to return multiple rows. List contains the Map, Map contains the key, value pairs. One Map contains one Row and One key, value pair contains one column. Key is the column name and value is the column value. How many no. of rows will be there that many no. of Maps will be created. If there is any problem executing the query it throws **DataAccessException**

public List queryForList(String sql, Class elementType):-

This method is similar to above method and also it takes the required type of element in the result list (for example, Integer.class). Here it throws **org.springframework.jdbc.UncategorizedSQLException** if we give different type.

And also throws **org.springframework.jdbc.IncorrectResultSetColumnCountException.** It returns a List of objects that match the specified element type.

List queryForList(String sql, Object[] args):-

This method will take query and also Object[] which takes the values for placeholders. It is same as above method.

List queryForList(String sql, Object[] args, Class elementType):-

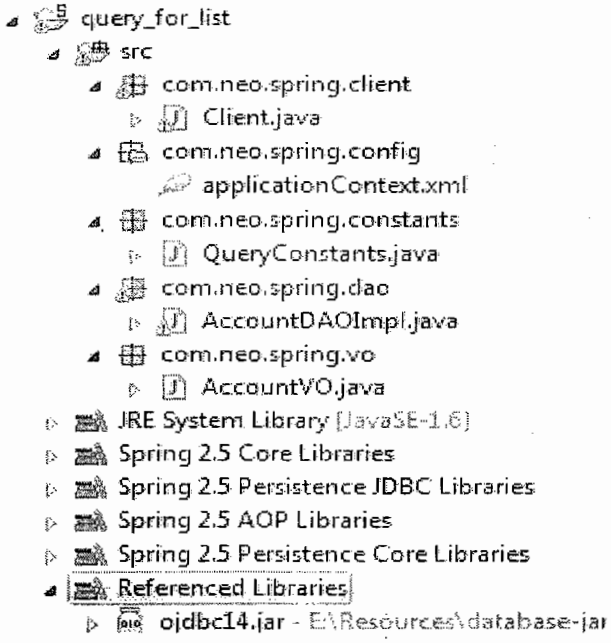
This method is similar to above methods.

List queryForList(String sql, Object[] args, int[] argTypes):-

This method is also similar to above method and it has int[] which takes the types of place holders.

List queryForList(String sql, Object[] args, int[] argTypes, Class elementType):-

It is also similar to above methods.



applicationContext.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3.     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xmlns:p="http://www.springframework.org/schema/p"
5.     xsi:schemaLocation="http://www.springframework.org/schema/beans
6. http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
7.
8. <!--  Apache datasource configuration -->
9.     <bean id="apacheDatasourceRef"
10.                          class="org.apache.commons.dbcp.BasicDataSource">
11.      <property name="driverClassName"
12.                                value="oracle.jdbc.driver.OracleDriver" />
13.      <property name="url" value="jdbc:oracle:thin:@localhost:1521:XE" />
14.      <property name="username" value="system" />
15.      <property name="password" value="tiger" />
16.    </bean>
17.
18.    <bean id="accountDAORef" class="com.neo.spring.dao.AccountDAOImpl">
19.          <property name="dataSource" ref=" apacheDatasourceRef " />
20.    </bean>
21.
22.    </beans>
```

QueryConstants.java

```
1. package com.neo.spring.constants;
2. public interface QueryConstants {
3.     public static String SELECT_QUERY_001 ="SELECT COUNT(*) FROM ACCOUNT" ;
4.     public static String SELECT_QUERY_002 = "SELECT NAME FROM ACCOUNT WHERE
5.                                                          ACCNO=1005" ;
6.     public static String SELECT_QUERY_003 = "SELECT NAME FROM ACCOUNT WHERE
7.                                                            ACCNO=?" ;
8.     public static String SELECT_QUERY_004 = "SELECT BAL FROM ACCOUNT WHERE
9.                                                          ACCNO=?" ;
```

```
10.   public static String SELECT_QUERY_005 = "SELECT NAME FROM ACCOUNT" ;
11.   public static String SELECT_QUERY_006 = "SELECT BAL FROM ACCOUNT" ;
12.   public static String SELECT_QUERY_007 = "SELECT NAME FROM ACCOUNT WHERE
13.                                                                 BAL>?" ;
14.   public static String SELECT_QUERY_008 = "SELECT BAL FROM ACCOUNT WHERE
15.                                                                 BAL>?" ;
16.   public static String SELECT_QUERY_009 = "SELECT ACCNO, NAME, BAL FROM
17.                                             ACCOUNT WHERE ACCNO=1005" ;
18.   public static String SELECT_QUERY_010 = "SELECT ACCNO, NAME, BAL FROM
19.                                                               ACCOUNT" ;
20.   public static String SELECT_QUERY_011 = "SELECT * FROM ACCOUNT" ;
21.   public static String SELECT_QUERY_012 = "SELECT NAME FROM ACCOUNT" ;
22.   public static String SELECT_QUERY_013 ="SELECT NAME,BAL FROM ACCOUNT" ;
23.
24.   }
```

AccountVO.java

```
1. package com.neo.spring.vo;
2. public class AccountVO {
3.     private int accno;
4.     private String name;
5.     private double balance;
6.
7.        public AccountVO(){
8.        }
9.
10.       public AccountVO(int accno, String name, double balance) {
11.               this.accno = accno;
12.               this.name = name;
13.               this.balance = balance;
14.          }
15.          //setters & getters
16.
17.   }
```

AccountDAOImpl.java

```
1. package com.neo.spring.dao;
2.
3. import java.sql.Types;
4. import java.util.List;
5.
6. import org.springframework.jdbc.core.support.JdbcDaoSupport;
7. import com.neo.spring.constants.QueryConstants;
8.
9. public class AccountDAOImpl extends JdbcDaoSupport{
10.
11.          public List usingQueryForList001(){
12.                return getJdbcTemplate().queryForList(
13.                                        QueryConstants.SELECT_QUERY_001);
14.          }
15.
16.          public List usingQueryForList002(){
17.                return getJdbcTemplate().queryForList(
18.                                        QueryConstants.SELECT_QUERY_002);
19.          }
```

```
20.
21.         public List usingQueryForList003(){
22.               return getJdbcTemplate().queryForList(
23.                 QueryConstants.SELECT_QUERY_003,new Object[]{1005});
24.         }
25.
26.         public List usingQueryForList004(){
27.               return getJdbcTemplate().queryForList(
28.                       QueryConstants.SELECT_QUERY_004,new Object[]{1005},
29.                                                 new int[]{Types.INTEGER});
30.         }
31.
32.         public List usingQueryForList005(){
33.               return getJdbcTemplate().queryForList(
34.                                    QueryConstants.SELECT_QUERY_005);
35.         }
36.
37.         public List usingQueryForList006(){
38.               return getJdbcTemplate().queryForList(
39.                      QueryConstants.SELECT_QUERY_006, Double.class);
40.         }
41.
42.         public List usingQueryForList007(){
43.               return getJdbcTemplate().queryForList(
44.                       QueryConstants.SELECT_QUERY_007,new Object[]{100},
45.                                                          String.class);
46.         }
47.
48.         public List usingQueryForList008(){
49.               return getJdbcTemplate().queryForList(
50.                       QueryConstants.SELECT_QUERY_008,new Object[]{100},
51.                                    new int[]{Types.DOUBLE}, Double.class);
52.         }
53.
54.         public List usingQueryForList009(){
55.               return getJdbcTemplate().queryForList(
56.                                          QueryConstants.SELECT_QUERY_009);
57.         }
58.
59.         public List usingQueryForList010(){
60.         return getJdbcTemplate().queryForList(
61.                                    QueryConstants.SELECT_QUERY_010);
62.         }
63.
64.         public List usingQueryForList011(){
65.               return getJdbcTemplate().queryForList(
66.                                          QueryConstants.SELECT_QUERY_011);
67.         }
68.
69.         public List usingQueryForList012(){
70.               return getJdbcTemplate().queryForList(
71.                           QueryConstants.SELECT_QUERY_012, Integer.class);
72.         }
73.
74.         public List usingQueryForList013(){
```

```
75.             return getJdbcTemplate().queryForList(
76.                         QueryConstants.SELECT_QUERY_013, String.class);
77.         }
78.   }
```

Client.java

```
1. package com.neo.spring.client;
2.
3. import java.util.List;
4.
5. import org.springframework.context.ApplicationContext;
6. import org.springframework.context.support.ClassPathXmlApplicationContext;
7.
8. import com.neo.spring.dao.AccountDAOImpl;
9.
10.   public class Client {
11.         private static ApplicationContext context = new
12.                 ClassPathXmlApplicationContext(
13.                     "com/neo/spring/config/applicationContext.xml");
14.
15.         public static void main(String[] args) {
16.               AccountDAOImpl impl = (AccountDAOImpl)
17.                                     context.getBean("accountDAORef");
18.
19.               sop("\n---usingQueryForList001---");
20.               List list = impl.usingQueryForList001();
21.               sop(list);
22.
23.               sop("\n---usingQueryForList002---");
24.               list = impl.usingQueryForList002();
25.               sop(list);
26.
27.               sop("\n---usingQueryForList003---");
28.               list = impl.usingQueryForList003();
29.               sop(list);
30.
31.               sop("\n---usingQueryForList004---");
32.               list = impl.usingQueryForList004();
33.               sop(list);
34.
35.
36.               sop("\n---usingQueryForList005---");
37.               list = impl.usingQueryForList005();
38.               sop(list);
39.
40.               sop("\n---usingQueryForList006---");
41.               list = impl.usingQueryForList006();
42.               sop(list);
43.
44.               sop("\n---usingQueryForList007---");
45.               list = impl.usingQueryForList007();
46.               sop(list);
47.
48.               sop("\n---usingQueryForList008---");
49.               list = impl.usingQueryForList008();
```

```
50.                 sop(list);
51.
52.
53.                 sop("\n---usingQueryForList009---");
54.                 list = impl.usingQueryForList009();
55.                 sop(list);
56.
57.                 sop("\n---usingQueryForList010---");
58.                 list = impl.usingQueryForList010();
59.
60.                 sop("\n---usingQueryForList011---");
61.                 list = impl.usingQueryForList011();
62.
63.   //            sop("\n---usingQueryForList012---");
64.   //            impl.usingQueryForList012();
65.                 /*UncategorizedSQLException*/
66.
67.   //            sop("\n---usingQueryForList013---");
68.   //            impl.usingQueryForList013();
69.                 /*IncorrectResultSetColumnCountException*/
70.
71.         }
72.
73.         public static void sop(Object object) {
74.                 System.out.println(object);
75.         }
76.   }
```

Before execution Account table

ACCNO	NAME	BAL
1004	somu	9800
1005	sekharreddy	2490
1003	somasekhar	7890

query()

It is not possible to send the ResultSet directly to the presentation layer, because ResultSet is always associated with database connection, which is costly affair. So we have to use DTO object which is used to transfer the data among different layers.
And also it is not possible to type cast ResultSet object to our DTO (Account) object.
To overcome these problem we can go for RowMapper interface.

RowMapper:-

RowMapper is an interface and it has one method **public Object mapRow(ResultSet rs, int rowNum)** method.

Where we can map the ResultSet data to our domain object(Account).

If we want to use RowMapper, We have to implement this RowMapper and override this mapRow method. Here we can't perform any iteration.

ResultSetExtractor:-

ResultSetExtractor is an interface and it has **public Object extractData(ResultSet rs)** method. It is similar to RowMapper. It can overcome the problem in RowMapper i.e., here we have the control on process of ResultSet extraction according to our requirement. Here we can perform looping according to our requirement.

RollCallbackHandler:-

RollCallbackHandler is an interface and it has one method **public void processRow(ResultSet rs).** It is also similar to ResultSetExtractor. Here also we have the control on extracting the ResultSet. But it returns nothing.

Difference between ResultSetExtractor and RowCallbackHandler:-

In case of ResultSetExtract, whatever extractData() returning i.e., return to query() method of jdbcTemplate then it inturn the same to the caller.
Whereas In case of RowCallbackHandler, it's processRow() method returns "void" to query() method of jdbcTemplate then it inturn the same to the caller.

Q) When to go for RowMapper?

Ans: If we don't want to control on extracting the ResultSet then we can go for RowMapper.

Q) When to go for ResultSetExtractor?

Ans: If we want to have the control on processing the ResultSet then we can go for ResultSetExtractor.

Q) When to go for RowCallbackHandler?

Ans: If we are not expecting any data after calling a method but we need some process has to be performed and later if we want to get the processed data then by using getter method we can access it. Then for this requirement we can go for RowCallbackHandler.

Q) What is Anonymous class?

Ans: A class which doesn't have any name is called as Anonymous class.

Q) What is the use of Anonymous class?
Ans: Let us consider the following example:

```
public interface I{
      x();
}
```

```
public class Impl implements I{
   public void x(){
         ------
         ------
   }
}
```

```
public class A{
   public void abc(I i){
         ------
         ------
   }
}
```

```
Impl impl=new Impl();
A a=new A();
a.abc(impl);
```

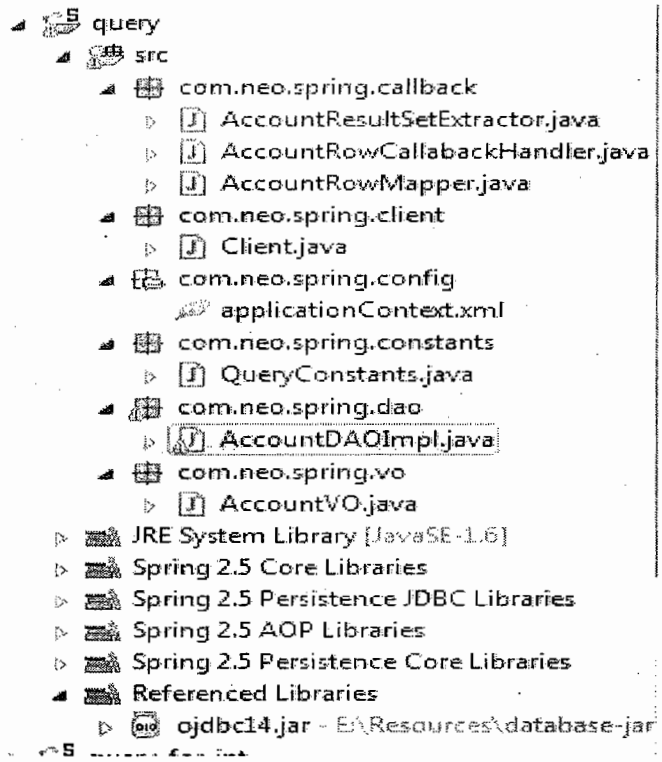
It is general approach and it is useful when we want to reuse the Impl class in several times.
If we don't have the requirement of reusing the implementation of an interface, the we can go for Anonymous class.

```
public interface I{
        x();
}
```

```
public class A{
   public void abc(I i){
                ------
                ------
    }
}
```

```
A a=new A();
a.abc(new I(){
            public void x(){
                --------
                --------
              }});
}
```

If we observe the implementation for the interface is provided as an argument to that method and the implementation is not having any name. So we can call it as an Anonymous class. Here instead of creating another class we can pass the implementation directly by using Anonymous class.



applicationContext.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3.     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xmlns:p="http://www.springframework.org/schema/p"
5.     xsi:schemaLocation="http://www.springframework.org/schema/beans
6. http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
7.
8. <!--  Apache datasource configuration -->
9.     <bean id="apacheDatasourceRef"
10.                          class="org.apache.commons.dbcp.BasicDataSource">
11.      <property name="driverClassName"
12.                                  value="oracle.jdbc.driver.OracleDriver" />
13.      <property name="url" value="jdbc:oracle:thin:@localhost:1521:XE" />
14.      <property name="username" value="system" />
15.      <property name="password" value="tiger" />
```

```
16.   </bean>
17.
18.   <bean id="accountDAORef" class="com.neo.spring.dao.AccountDAOImpl">
19.         <property name="dataSource" ref=" apacheDatasourceRef " />
20.   </bean>
21.
22.   </beans>
```

QueryConstants.java

```
1. package com.neo.spring.constants;
2.
3. public interface QueryConstants {
4.    public static String SELECT_QUERY_001 = "SELECT *FROM ACCOUNT WHERE
5.                                                          ACCNO=1005";
6.    public static String SELECT_QUERY_002 = "SELECT *FROM ACCOUNT WHERE
7.                                                          BAL>?";
8. }
```

AccountVO.java

```
1. package com.neo.spring.vo;
2. public class AccountVO {
3.    private int accno;
4.    private String name;
5.    private double balance;
6.
7.    public AccountVO(){
8.    }
9.
10.   public AccountVO(int accno, String name, double balance) {
11.         this.accno = accno;
12.         this.name = name;
13.         this.balance = balance;
14.   }
15.   //setters & getters
16.
17.   @Override
18.   public String toString() {
19.         return "ACCNO :"+accno+" || NAME :"+name+" ||BALANCE :"+balance;
20.         }
21.
22.   }
```

AccountDAOImpl.java

```
1. package com.neo.spring.dao;
2.
3.
4. import java.sql.ResultSet;
5. import java.sql.SQLException;
6. import java.util.ArrayList;
7. import java.util.List;
8.
9. import org.springframework.dao.DataAccessException;
10.   import org.springframework.jdbc.core.ResultSetExtractor;
11.   import org.springframework.jdbc.core.RowCallbackHandler;
12.   import org.springframework.jdbc.core.RowMapper;
```

```
13.  import org.springframework.jdbc.core.support.JdbcDaoSupport;
14.
15.  import com.neo.spring.callback.AccountResultSetExtractor;
16.  import com.neo.spring.callback.AccountRowCallabackHandler;
17.  import com.neo.spring.callback.AccountRowMapper;
18.  import com.neo.spring.constants.QueryConstants;
19.  import com.neo.spring.vo.AccountVO;
20.
21.  public class AccountDAOImpl extends JdbcDaoSupport{
22.        public AccountVO usingQuery001(){
23.              List<AccountVO> accountVOs=  getJdbcTemplate().query(
24.                QueryConstants.SELECT_QUERY_001, new AccountRowMapper());
25.              return accountVOs.get(0);
26.        }
27.
28.        public AccountVO usingQuery002(){
29.           List<AccountVO> accountVOs=  getJdbcTemplate().query(
30.           QueryConstants.SELECT_QUERY_001, new AccountRowMapperInner());
31.
32.           return accountVOs.get(0);
33.        }
34.
35.        public AccountVO usingQuery003(){
36.              List<AccountVO> accountVOs=  getJdbcTemplate().query(
37.                  QueryConstants.SELECT_QUERY_001,
38.                  new RowMapper() {
39.
40.                    @Override
41.                    public Object mapRow(ResultSet rs, int rowCount)
42.                                                throws SQLException {
43.                    return new AccountVO(rs.getInt(1),
44.                                    rs.getString(2), rs.getDouble(3));
45.                    }
46.              });
47.
48.               return accountVOs.get(0);
49.        }
50.
51.        public AccountVO usingQuery004(){
52.              List<AccountVO> accountVOs=  (List<AccountVO>)
53.               getJdbcTemplate().query(QueryConstants.SELECT_QUERY_001,
54.                                        new AccountResultSetExtractor());
55.              return accountVOs.get(0);
56.        }
57.
58.        public AccountVO usingQuery005(){
59.              List<AccountVO> accountVOs=  (List<AccountVO>)
60.                 getJdbcTemplate().query(QueryConstants.SELECT_QUERY_001,
61.                                    new AccountResultSetExtractorInner());
62.              return accountVOs.get(0);
63.        }
64.
65.        public AccountVO usingQuery006(){
66.              List<AccountVO> accountVOs=  (List<AccountVO>)
67.                 getJdbcTemplate().query(QueryConstants.SELECT_QUERY_001,
```

```
68.                 new ResultSetExtractor() {
69.
70.                     @Override
71.                     public Object extractData(ResultSet rs) throws
72.                                     SQLException, DataAccessException {
73.                        List<AccountVO> accountVOs=new
74.                                                 ArrayList<AccountVO>();
75.
76.                           while(rs.next()){
77.                             accountVOs.add(new AccountVO(rs.getInt(1),
78.                                       rs.getString(2), rs.getDouble(3)));
79.                           }
80.                           return accountVOs;
81.                     }
82.             });
83.             return accountVOs.get(0);
84.         }
85.
86.
87.         public void usingQuery007(){
88.             getJdbcTemplate().query(QueryConstants.SELECT_QUERY_001,
89.                                       new AccountRowCallabackHandler());
90.         }
91.         public void usingQuery008(){
92.             getJdbcTemplate().query(QueryConstants.SELECT_QUERY_001,
93.                                     new AccountRowCallabackHandler());
94.         }
95.
96.         public void usingQuery009(){
97.             getJdbcTemplate().query(QueryConstants.SELECT_QUERY_001,
98.             new RowCallbackHandler() {
99.
100.             @Override
101.             public void processRow(ResultSet rs) throws SQLException {
102.                 String name = rs.getString(2);
103.                 if(name.equals("sekharreddy")){
104.                    System.out.println("Trainer is executing ");
105.                 }else{
106.                    System.out.println("Student is executing ");
107.                 }
108.             }
109.             });
110.         }
111.
112.         public List<AccountVO> usingQuery010(){
113.             return getJdbcTemplate().query(
114.                 QueryConstants.SELECT_QUERY_002, new Object[]{105.0} ,
115.                                           new AccountRowMapperInner());
116.         }
117.
118.         public List<AccountVO> usingQuery011(){
119.             return (List<AccountVO>)getJdbcTemplate().query(
120.               QueryConstants.SELECT_QUERY_002, new Object[]{105.0} ,
121.                                     new AccountResultSetExtractorInner());
122.         }
```

```
123.
124.        private class AccountRowCallbackHandler implements
125.                                                    RowCallbackHandler{
126.              @Override
127.              public void processRow(ResultSet rs) throws SQLException {
128.                    String name = rs.getString(2);
129.                    if(name.equals("sekharreddy")){
130.                          System.out.println("Trainer is executing");
131.                    }else{
132.                          System.out.println("Student is executing");
133.                    }
134.              }
135.        }
136.        private class AccountRowMapperInner implements RowMapper{
137.              @Override
138.              public Object mapRow(ResultSet rs, int rowCount) throws
139.                                                          SQLException {
140.                    return new AccountVO(rs.getInt(1), rs.getString(2),
141.                                                          rs.getDouble(3));
142.              }
143.        }
144.
145.        private class AccountResultSetExtractorInner implements
146.                                                    ResultSetExtractor{
147.              @Override
148.              public Object extractData(ResultSet rs) throws
149.                                    SQLException,DataAccessException {
150.                    List<AccountVO> accountVOs = new
151.                                                ArrayList<AccountVO>();
152.
153.                    while(rs.next()){
154.                          accountVOs.add(new AccountVO(rs.getInt(1),
155.                                rs.getString(2), rs.getDouble(3)));
156.                    }
157.                    return accountVOs;
158.              }
159.        }
160.
161.
162.
163.  }
```

Client.java

```
1. package com.neo.spring.client;
2. import java.util.List;
3.
4. import org.springframework.context.ApplicationContext;
5. import org.springframework.context.support.ClassPathXmlApplicationContext;
6.
7. import com.neo.spring.dao.AccountDAOImpl;
8. import com.neo.spring.vo.AccountVO;
9.
10.  public class Client {
11.        private static ApplicationContext context = new
12.              ClassPathXmlApplicationContext(
```

```
13.                   "com/neo/spring/config/applicationContext.xml");
14.
15.       public static void main(String[] args) {
16.             AccountDAOImpl impl= (AccountDAOImpl)
17.                                   context.getBean("accountDAORef");
18.
19.             sop("\n---usingQuery001---");
20.             AccountVO accountVO = impl.usingQuery001();
21.             sop("Account details are ....");
22.             sop(accountVO);
23.
24.             sop("\n---usingQuery002---");
25.             accountVO = impl.usingQuery002();
26.             sop("Account details are ....");
27.             sop(accountVO);
28.
29.             sop("\n---usingQuery003---");
30.             accountVO = impl.usingQuery003();
31.             sop("Account details are ....");
32.             sop(accountVO);
33.
34.             sop("\n---usingQuery004---");
35.             accountVO = impl.usingQuery004();
36.             sop("Account details are ....");
37.             sop(accountVO);
38.
39.             sop("\n---usingQuery005---");
40.             accountVO = impl.usingQuery005();
41.             sop("Account details are ....");
42.             sop(accountVO);
43.
44.             sop("\n---usingQuery006---");
45.             accountVO = impl.usingQuery006();
46.             sop("Account details are ....");
47.             sop(accountVO);
48.
49.             sop("\n---usingQuery007---");
50.             impl.usingQuery007();
51.
52.             sop("\n---usingQuery008---");
53.             impl.usingQuery008();
54.
55.             sop("\n---usingQuery009---");
56.             impl.usingQuery009();
57.
58.             sop("\n---usingQuery010---");
59.             List<AccountVO> accountVOs = impl.usingQuery010();
60.             for(AccountVO vo : accountVOs)
61.                   sop(vo);
62.
63.             sop("\n---usingQuery011---");
64.             accountVOs = impl.usingQuery011();
65.             for(AccountVO vo : accountVOs)
66.                   sop(vo);
67.       }
```

```
68.
69.         public static void sop(Object object) {
70.               System.out.println(object);
71.         }
72.   }
```

AccountResultSetExtractor.java

```
1. package com.neo.spring.callback;
2.
3. import java.sql.ResultSet;
4. import java.sql.SQLException;
5. import java.util.ArrayList;
6. import java.util.List;
7.
8. import org.springframework.dao.DataAccessException;
9. import org.springframework.jdbc.core.ResultSetExtractor;
10.
11.   import com.neo.spring.vo.AccountVO;
12.
13.   public class AccountResultSetExtractor implements ResultSetExtractor{
14.   @Override
15.   public Object extractData(ResultSet rs) throws SQLException,
16.               DataAccessException {
17.         List<AccountVO> accountVOs = new ArrayList<AccountVO>();
18.
19.         while(rs.next()){
20.               accountVOs.add(new AccountVO(rs.getInt(1), rs.getString(2),
21.                                                   rs.getDouble(3)));
22.         }
23.         return accountVOs;
24.   }
25.   }
```

AccountRowCallabackHandler.java.java

```
1. package com.neo.spring.callback;
2.
3. import java.sql.ResultSet;
4. import java.sql.SQLException;
5.
6. import org.springframework.jdbc.core.RowCallbackHandler;
7.
8. public class AccountRowCallabackHandler implements RowCallbackHandler {
9.    @Override
10.         public void processRow(ResultSet rs) throws SQLException {
11.               String name = rs.getString(2);
12.               if(name.equals("sekharreddy")){
13.                     System.out.println("Trainer is executing");
14.               }else{
15.                     System.out.println("Student is executing");
16.               }
17.
18.         }
19.   }
```

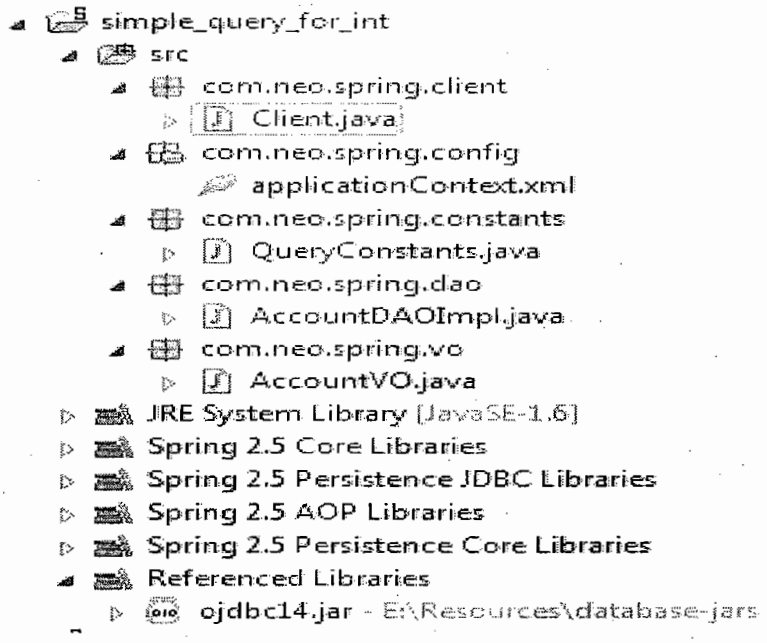
AccountRowMapper.java

```
1. package com.neo.spring.callback;
2.
3. import java.sql.ResultSet;
4. import java.sql.SQLException;
5.
6. import org.springframework.jdbc.core.RowMapper;
7.
8. import com.neo.spring.vo.AccountVO;
9.
10.   public class AccountRowMapper implements RowMapper {
11.         @Override
12.         public Object mapRow(ResultSet rs, int rc) throws SQLException {
13.               return new AccountVO(rs.getInt(1), rs.getString(2),
14.                                                   rs.getDouble(3));
15.         }
16.   }
```

Before execution Account table

ACCNO	NAME	BAL
1004	somu	9800
1005	sekharreddy	2490
1003	somasekhar	7890

Q.)Develop applications where we can use SimpleJdbcTemplate's queryForInt(...), queryForObject(...), queryForMap(...), queryForList(...), query(...) methods.



applicationContext.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3.    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xmlns:p="http://www.springframework.org/schema/p"
5.    xsi:schemaLocation="http://www.springframework.org/schema/beans
6. http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
7.
8. <!--  Apache datasource configuration -->
9.    <bean id="apacheDatasourceRef"
10.                        class="org.apache.commons.dbcp.BasicDataSource">
11.     <property name="driverClassName"
12.                              value="oracle.jdbc.driver.OracleDriver" />
13.     <property name="url" value="jdbc:oracle:thin:@localhost:1521:XE" />
14.     <property name="username" value="system" />
15.     <property name="password" value="tiger" />
16.   </bean>
17.
18.   <bean id="accountDAORef" class="com.neo.spring.dao.AccountDAOImpl">
19.         <property name="dataSource" ref=" apacheDatasourceRef " />
20.   </bean>
21.
22.   </beans>
```

AccountVO.java

```
1. package com.neo.spring.vo;
2. public class AccountVO {
3.    private int accno;
4.    private String name;
5.    private double balance;
```

```
6.
7.    public AccountVO(){
8.    }
9.
10.   public AccountVO(int accno, String name, double balance) {
11.         this.accno = accno;
12.         this.name = name;
13.         this.balance = balance;
14.   }
15.   //setters & getters
16.
17.   @Override
18.   public String toString() {
19.         return "ACCNO :"+accno+" || NAME :"+name+" ||BALANCE :"+balance;
20.         }
21.
22.   }
```

QueryConstants.java

```
1. package com.neo.spring.constants;
2.
3. public interface QueryConstants {
4.    public static String SELECT_QUERY_001 = "SELECT COUNT(*) FROM ACCOUNT
5.                                             WHERE BAL>? AND NAME LIKE ?";
6.    public static String SELECT_QUERY_002 = "SELECT COUNT(*) FROM ACCOUNT
7.                                          WHERE BAL>:bal AND NAME LIKE :name";
8.
9. }
```

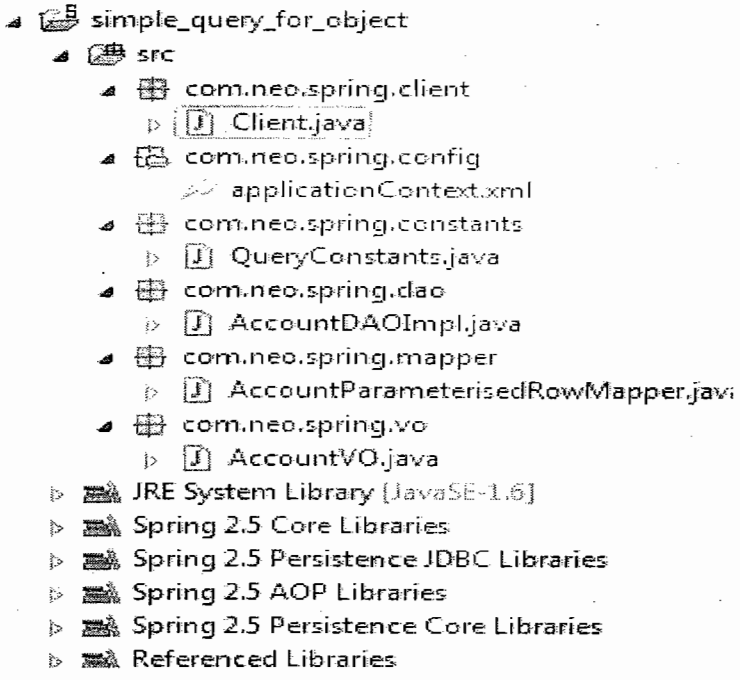
AccountDAOImpl.java

```
1. package com.neo.spring.dao;
2.
3. import java.util.HashMap;
4. import java.util.Map;
5.
6. import org.springframework.jdbc.core.namedparam.MapSqlParameterSource;
7. import org.springframework.jdbc.core.namedparam.SqlParameterSource;
8. import org.springframework.jdbc.core.simple.SimpleJdbcDaoSupport;
9.
10.   import com.neo.spring.constants.QueryConstants;
11.
12.   public class AccountDAOImpl extends SimpleJdbcDaoSupport{
13.
14.         public int usingQueryForInt001(){
15.               return getSimpleJdbcTemplate().queryForInt(
16.                     QueryConstants.SELECT_QUERY_001,1000.0,"%sekhar%");
17.         }
18.         public int usingQueryForInt002(){
19.               Map<String,Object> map=new HashMap<String,Object>();
20.               map.put("bal",1000.0);
21.               map.put("name","%sekhar%");
22.               return getSimpleJdbcTemplate().queryForInt(
23.                                    QueryConstants.SELECT_QUERY_002,map);
24.         }
25.         public int usingQueryForInt003(){
```

```
26.             Map<String,Object> map=new HashMap<String,Object>();
27.             map.put("bal",1000.0);
28.             map.put("name","%sekhar%");
29.
30.             SqlParameterSource params=new MapSqlParameterSource(map);
31.             return getSimpleJdbcTemplate().queryForInt(
32.                             QueryConstants.SELECT_QUERY_002,params);
33.         }
34.
35.     }
```

Client.java

```
1. package com.neo.spring.client;
2.
3. import org.springframework.context.ApplicationContext;
4. import org.springframework.context.support.ClassPathXmlApplicationContext;
5.
6. import com.neo.spring.dao.AccountDAOImpl;
7.
8. public class Client {
9.    private static ApplicationContext context = new
10.             ClassPathXmlApplicationContext(
11.                   "com/neo/spring/config/applicationContext.xml");
12.
13.         public static void main(String[] args) {
14.             AccountDAOImpl impl= (AccountDAOImpl)
15.                                 context.getBean("accountDAORef");
16.
17.             sop("\n---usingQueryForInt001---");
18.             int rowCount = impl.usingQueryForInt001();
19.             sop("No.of rows :"+rowCount);
20.
21.             sop("\n---usingQueryForInt002---");
22.             rowCount = impl.usingQueryForInt002();
23.             sop("No.of rows :"+rowCount);
24.
25.             sop("\n---usingQueryForInt003---");
26.             rowCount = impl.usingQueryForInt003();
27.             sop("No.of rows :"+rowCount);
28.
29.         }
30.
31.         public static void sop(Object object) {
32.             System.out.println(object);
33.         }
34.   }
```



applicationContext.xml

```
<!-SAME AS ABOVE-->
```

AccountVO.java

```
<!-SAME AS ABOVE-->
```

QueryConstants.java

```
1. package com.neo.spring.constants;
2.
3. public interface QueryConstants {
4.     public static String SELECT_QUERY_001 = "SELECT NAME FROM ACCOUNT WHERE
5.                                                          ACCNO = ?";
6.     public static String SELECT_QUERY_002 = "SELECT NAME FROM ACCOUNT WHERE
7.                                                        ACCNO = :accno";
8.     public static String SELECT_QUERY_003 = "SELECT ACCNO,NAME,BAL FROM
9.                                              ACCOUNT WHERE ACCNO = ?";
10.    public static String SELECT_QUERY_004 = "SELECT * FROM ACCOUNT WHERE
11.                                                        ACCNO = :acno";
12.    }
```

AccountParameterisedRowMapper.java

```
1. package com.neo.spring.mapper;
2.
3. import java.sql.ResultSet;
4. import java.sql.SQLException;
5.
6. import org.springframework.jdbc.core.simple.ParameterizedRowMapper;
7.
8. import com.neo.spring.vo.AccountVO;
9.
10.   public class AccountParameterisedRowMapper implements
11.                                   ParameterizedRowMapper<AccountVO>{
12.
```

```
13.         @Override
14.         public AccountVO mapRow(ResultSet rs, int rowCount) throws
15.                                                         SQLException {
16.               return new AccountVO(rs.getInt(1), rs.getString(2),
17.                                                   rs.getDouble(3));
18.         }
19.   }
```

AccountDAOImpl.java

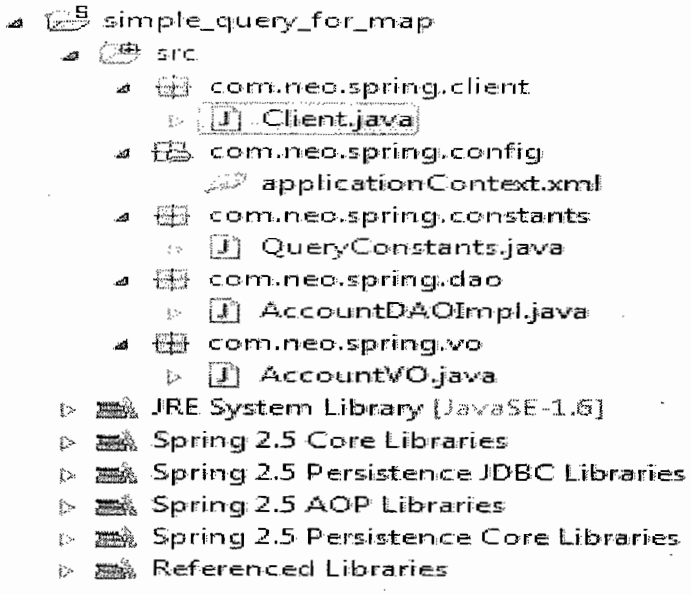
```
1. package com.neo.spring.dao;
2.
3. import java.util.HashMap;
4. import java.util.Map;
5.
6. import org.springframework.jdbc.core.namedparam.MapSqlParameterSource;
7. import org.springframework.jdbc.core.namedparam.SqlParameterSource;
8. import org.springframework.jdbc.core.simple.SimpleJdbcDaoSupport;
9.
10.   import com.neo.spring.constants.QueryConstants;
11.   import com.neo.spring.mapper.AccountParameterisedRowMapper;
12.   import com.neo.spring.vo.AccountVO;
13.
14.   public class AccountDAOImpl extends SimpleJdbcDaoSupport{
15.         public String usingQueryForObject001(){
16.               return getSimpleJdbcTemplate().queryForObject(
17.                     QueryConstants.SELECT_QUERY_001, String.class,1005);
18.         }
19.
20.         public String usingQueryForObject002(){
21.               Map<String, Object> map=new HashMap<String, Object>();
22.               map.put("accno", 1005);
23.               return getSimpleJdbcTemplate().queryForObject(
24.                     QueryConstants.SELECT_QUERY_002, String.class,map);
25.         }
26.         public String usingQueryForObject003(){
27.               Map<String, Object> map=new HashMap<String, Object>();
28.               map.put("accno", 1005);
29.               SqlParameterSource params=new MapSqlParameterSource(map);
30.               return getSimpleJdbcTemplate().queryForObject(
31.                  QueryConstants.SELECT_QUERY_002, String.class,params);
32.         }
33.
34.         public AccountVO usingQueryForObject004(){
35.               return getSimpleJdbcTemplate().queryForObject(
36.                     QueryConstants.SELECT_QUERY_003,
37.                           new AccountParameterisedRowMapper(),1005);
38.         }
39.
40.         public AccountVO usingQueryForObject005(){
41.               Map<String, Object> map=new HashMap<String, Object>();
42.               map.put("acno", 1005);
43.               return getSimpleJdbcTemplate().queryForObject(
44.                     QueryConstants.SELECT_QUERY_004,
45.                           new AccountParameterisedRowMapper(),map);
46.         }
```

```
47.         public AccountVO usingQueryForObject006(){
48.               Map<String, Object> map=new HashMap<String, Object>();
49.               map.put("acno", 1005);
50.               SqlParameterSource params=new MapSqlParameterSource(map);
51.               return getSimpleJdbcTemplate().queryForObject(
52.                     QueryConstants.SELECT_QUERY_004,
53.                           new AccountParameterisedRowMapper(),params);
54.         }
55.
56.   }
```

Client.java

```
1. package com.neo.spring.client;
2.
3. import org.springframework.context.ApplicationContext;
4. import org.springframework.context.support.ClassPathXmlApplicationContext;
5.
6. import com.neo.spring.dao.AccountDAOImpl;
7. import com.neo.spring.vo.AccountVO;
8.
9. public class Client {
10.         private static ApplicationContext context = new
11.                 ClassPathXmlApplicationContext(
12.                     "com/neo/spring/config/applicationContext.xml");
13.
14.         public static void main(String[] args) {
15.               AccountDAOImpl impl= (AccountDAOImpl)
16.                                       context.getBean("accountDAORef");
17.
18.               sop("\n---usingQueryForObject001---");
19.               String name = impl.usingQueryForObject001();
20.               sop("Name :"+name);
21.
22.               sop("\n---usingQueryForObject002---");
23.               name = impl.usingQueryForObject002();
24.               sop("Name :"+name);
25.
26.               sop("\n---usingQueryForObject003---");
27.               name = impl.usingQueryForObject003();
28.               sop("Name :"+name);
29.
30.               sop("\n---usingQueryForObject004---");
31.               AccountVO vo = impl.usingQueryForObject004();
32.               sop(vo);
33.
34.               sop("\n---usingQueryForObject005---");
35.                vo = impl.usingQueryForObject005();
36.               sop(vo);
37.
38.               sop("\n---usingQueryForObject006---");
39.               vo = impl.usingQueryForObject006();
40.               sop(vo);
41.   }
42.
43.         public static void sop(Object object) {
```

```
44.                 System.out.println(object);
45.         }
46.
47.   }
```



applicationContext.xml

```
<!-SAME AS ABOVE-->
```

AccountVO.java

```
<!-SAME AS ABOVE-->
```

QueryConstants.java

```
1. package com.neo.spring.constants;
2.
3. public interface QueryConstants {
4.     public static String SELECT_QUERY_001 = "SELECT NAME FROM ACCOUNT WHERE
5.                                                                 ACCNO=?" ;
6.     public static String SELECT_QUERY_002 = "SELECT * FROM ACCOUNT WHERE
7.                                                             ACCNO=:accno";
8. }
```

AccountDAOImpl.java

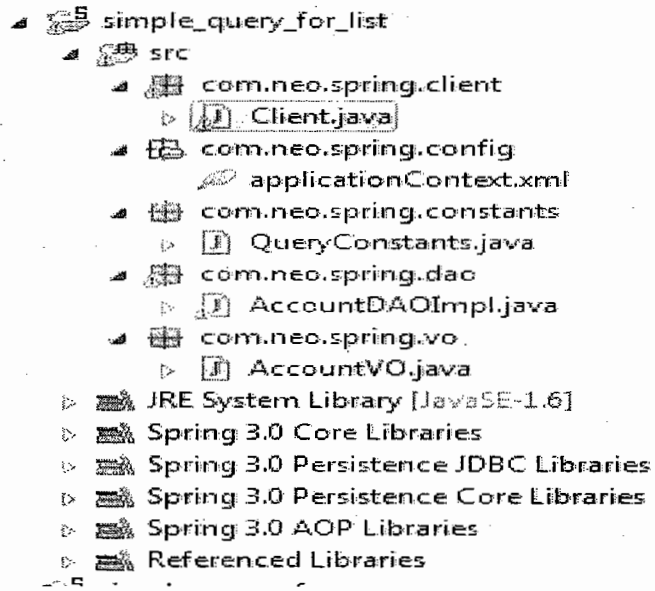
```
1. package com.neo.spring.dao;
2.
3. import java.util.HashMap;
4. import java.util.Map;
5.
6. import org.springframework.jdbc.core.namedparam.MapSqlParameterSource;
7. import org.springframework.jdbc.core.namedparam.SqlParameterSource;
8. import org.springframework.jdbc.core.simple.SimpleJdbcDaoSupport;
9.
10.    import com.neo.spring.constants.QueryConstants;
11.
12.    public class AccountDAOImpl extends SimpleJdbcDaoSupport{
13.
```

```
14.         public Map<String, Object> usingQueryForMap001(){
15.               return getSimpleJdbcTemplate().queryForMap(
16.                                    QueryConstants.SELECT_QUERY_001,1005);
17.         }
18.
19.         public Map<String, Object> usingQueryForMap002(){
20.               Map<String, Object> map=new HashMap<String, Object>();
21.               map.put("accno", 1005);
22.               return getSimpleJdbcTemplate().queryForMap(
23.                                     QueryConstants.SELECT_QUERY_002,map);
24.         }
25.
26.         public Map<String, Object> usingQueryForMap003(){
27.               Map<String, Object> map=new HashMap<String, Object>();
28.               map.put("accno", 1005);
29.               SqlParameterSource params=new MapSqlParameterSource(map);
30.               return getSimpleJdbcTemplate().queryForMap(
31.                                    QueryConstants.SELECT_QUERY_002,params);
32.         }
33.
34.   }
```

Client.java

```
1. package com.neo.spring.client;
2.
3. import java.util.Iterator;
4. import java.util.Map;
5. import java.util.Set;
6. import java.util.Map.Entry;
7.
8. import org.springframework.context.ApplicationContext;
9. import org.springframework.context.support.ClassPathXmlApplicationContext;
10.
11.   import com.neo.spring.dao.AccountDAOImpl;
12.
13.   public class Client {
14.         private static ApplicationContext context = new
15.                 ClassPathXmlApplicationContext(
16.                     "com/neo/spring/config/applicationContext.xml");
17.
18.         public static void main(String[] args) {
19.               AccountDAOImpl impl= (AccountDAOImpl)
20.                                       context.getBean("accountDAORef");
21.
22.               sop("\n---usingQueryForMap001---");
23.               Map<String, Object> map = impl.usingQueryForMap001();
24.               sop(map);
25.               Set<String> keys = map.keySet();
26.               Iterator<String> itr = keys.iterator();
27.               while(itr.hasNext()){
28.                     String key = itr.next();
29.                     sop(key+"==>"+map.get(key));
30.               }
31.
32.               sop("\n---usingQueryForMap002---");
```

```
33.                map= impl.usingQueryForMap002();
34.                for(String key : map.keySet())
35.                      sop(key+"==>"+map.get(key));
36.
37.                sop("\n---usingQueryForMap003---");
38.                map=impl.usingQueryForMap003();
39.                Set<Entry<String, Object>> entries = map.entrySet();
40.                for(Entry<String, Object> entry : entries)
41.                      sop(entry.getKey()+"==>"+entry.getValue());
42.
43.
44.          }
45.
46.          public static void sop(Object object) {
47.                System.out.println(object);
48.          }
49.
50.    }
```



applicationContext.xml

```
<!—SAME AS ABOVE-->
```

AccountVO.java

```
<!—SAME AS ABOVE-->
```

QueryConstants.java

```
1. package com.neo.spring.constants;
2.
3. public interface QueryConstants {
4.    public static String SELECT_QUERY_001 = "SELECT NAME FROM ACCOUNT WHERE
5.                                                                ACCNO=?" ;
6.    public static String SELECT_QUERY_002 = "SELECT * FROM ACCOUNT WHERE
7.                                                              ACCNO=:accno" ;
8.
9. }
```

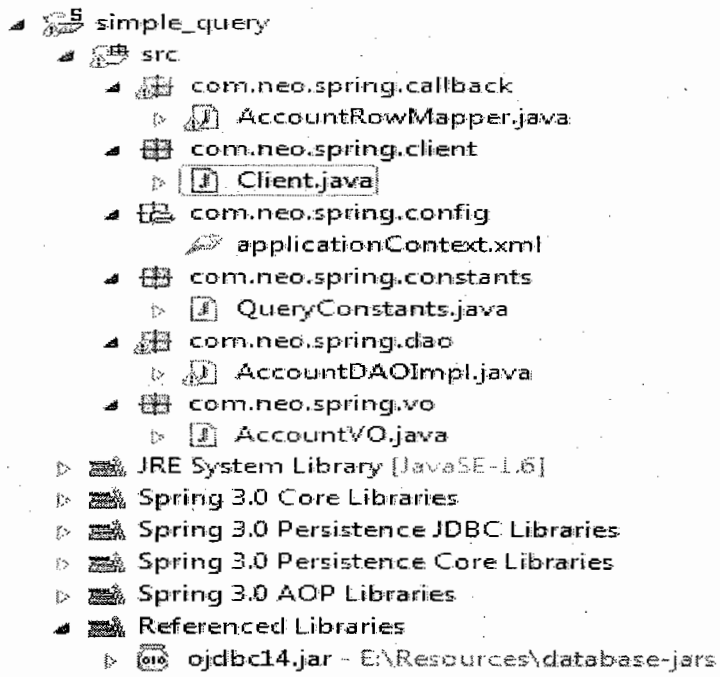
AccountDAOImpl.java

```
1. package com.neo.spring.dao;
2.
3. import java.util.HashMap;
4. import java.util.List;
5. import java.util.Map;
6.
7. import org.springframework.jdbc.core.namedparam.MapSqlParameterSource;
8. import org.springframework.jdbc.core.namedparam.SqlParameterSource;
9. import org.springframework.jdbc.core.simple.SimpleJdbcDaoSupport;
10.
11.   import com.neo.spring.constants.QueryConstants;
12.
13.   public class AccountDAOImpl extends SimpleJdbcDaoSupport{
14.
15.         public List usingQueryForList001(){
16.               return getSimpleJdbcTemplate().queryForList(
17.                                 QueryConstants.SELECT_QUERY_001,1005);
18.         }
19.         public List usingQueryForList002(){
20.               Map<String, Object> map=new HashMap<String, Object>();
21.               map.put("accno", 1005);
22.               return getSimpleJdbcTemplate().queryForList(
23.                                 QueryConstants.SELECT_QUERY_002,map);
24.         }
25.         public List usingQueryForList003(){
26.               Map<String, Object> map=new HashMap<String, Object>();
27.               map.put("accno", 1005);
28.               SqlParameterSource params=new MapSqlParameterSource(map);
29.               return getSimpleJdbcTemplate().queryForList(
30.                               QueryConstants.SELECT_QUERY_002,params);
31.         }
32.   }
```

Client.java

```
1. package com.neo.spring.client;
2.
3. import java.util.List;
4.
5. import org.springframework.context.ApplicationContext;
6. import org.springframework.context.support.ClassPathXmlApplicationContext;
7.
8. import com.neo.spring.dao.AccountDAOImpl;
9.
10.   public class Client {
11.         private static ApplicationContext context = new
12.               ClassPathXmlApplicationContext(
13.                     "com/neo/spring/config/applicationContext.xml");
14.
15.         public static void main(String[] args) {
16.               AccountDAOImpl impl = (AccountDAOImpl)
17.                                 context.getBean("accountDAORef");
18.
19.               sop("\n---usingQueryForList001---");
20.               List list = impl.usingQueryForList001();
```

```
21.                 sop(list);
22.
23.                 sop("\n---usingQueryForList002---");
24.                 list = impl.usingQueryForList002();
25.                 sop(list);
26.
27.                 sop("\n---usingQueryForList003---");
28.                 list = impl.usingQueryForList003();
29.                 sop(list);
30.           }
31.           public static void sop(Object object) {
32.                 System.out.println(object);
33.           }
34.
35.     }
```



applicationContext.xml

```
<!-SAME AS ABOVE-->
```

AccountVO.java

```
<!-SAME AS ABOVE-->
```

QueryConstants.java

```
1. package com.neo.spring.constants;
2.
3. public interface QueryConstants {
4.     public static String SELECT_QUERY_001 = "SELECT *FROM ACCOUNT WHERE
5.                                                                 BAL>?";
6.     public static String SELECT_QUERY_002 = "SELECT *FROM ACCOUNT WHERE
7.                                                                 BAL>:bal";
8. }
```

AccountRowMapper.java

```
1. package com.neo.spring.callback;
2.
3. import java.sql.ResultSet;
4. import java.sql.SQLException;
5.
6. import org.springframework.jdbc.core.RowMapper;
7.
8. import com.neo.spring.vo.AccountVO;
9.
10.   public class AccountRowMapper implements RowMapper {
11.         @Override
12.         public Object mapRow(ResultSet rs, int rc) throws SQLException {
13.               return new AccountVO(rs.getInt(1), rs.getString(2),
14.                                                   rs.getDouble(3));
15.         }
16.   }
```

AccountDAOImpl.java

```
1. package com.neo.spring.dao;
2.
3. import java.util.HashMap;
4. import java.util.List;
5. import java.util.Map;
6. import org.springframework.jdbc.core.namedparam.MapSqlParameterSource;
7. import org.springframework.jdbc.core.namedparam.SqlParameterSource;
8. import org.springframework.jdbc.core.simple.SimpleJdbcDaoSupport;
9. import com.neo.spring.callback.AccountRowMapper;
10.   import com.neo.spring.constants.QueryConstants;
11.   import com.neo.spring.vo.AccountVO;
12.
13.   public class AccountDAOImpl extends SimpleJdbcDaoSupport{
14.         public List<AccountVO>  usingQuery001(){
15.               return getSimpleJdbcTemplate().query(
16.                     QueryConstants.SELECT_QUERY_001,
17.                           new AccountRowMapper(), 105.0);
18.         }
19.
20.         public List<AccountVO>  usingQuery002(){
21.               Map<String, Object> map = new HashMap<String, Object>();
22.               map.put("bal", 105.0);
23.               return getSimpleJdbcTemplate().query(
24.                        QueryConstants.SELECT_QUERY_002,
25.                              new AccountRowMapper(), map);
26.         }
27.
28.         public List<AccountVO>  usingQuery003(){
29.               Map<String, Object> map = new HashMap<String, Object>();
30.               map.put("bal", 105.0);
31.               SqlParameterSource params = new MapSqlParameterSource(map);
32.               return getSimpleJdbcTemplate().query(
33.                     QueryConstants.SELECT_QUERY_002,
34.                           new AccountRowMapper(), params);
35.         }
36.   }
```

Client.java

```
1. package com.neo.spring.client;
2. import java.util.List;
3. import org.springframework.context.ApplicationContext;
4. import org.springframework.context.support.ClassPathXmlApplicationContext;
5. import com.neo.spring.dao.AccountDAOImpl;
6. import com.neo.spring.vo.AccountVO;
7.
8. public class Client {
9.    private static ApplicationContext context = new
10.          ClassPathXmlApplicationContext(
11.                    "com/neo/spring/config/applicationContext.xml");
12.
13.       public static void main(String[] args) {
14.             AccountDAOImpl impl= (AccountDAOImpl)
15.                                    context.getBean("accountDAORef");
16.
17.             sop("\n---usingQuery001---");
18.             List<AccountVO>  accountVOs = impl.usingQuery001();
19.             sop("Account details are ....");
20.             sop(accountVOs);
21.
22.             sop("\n---usingQuery002---");
23.             accountVOs = impl.usingQuery001();
24.             sop("Account details are ....");
25.             sop(accountVOs);
26.
27.             sop("\n---usingQuery003---");
28.             accountVOs = impl.usingQuery003();
29.             sop("Account details are ....");
30.             sop(accountVOs);
31.
32.       }
33.       public static void sop(Object object) {
34.             System.out.println(object);
35.       }
36.  }
```

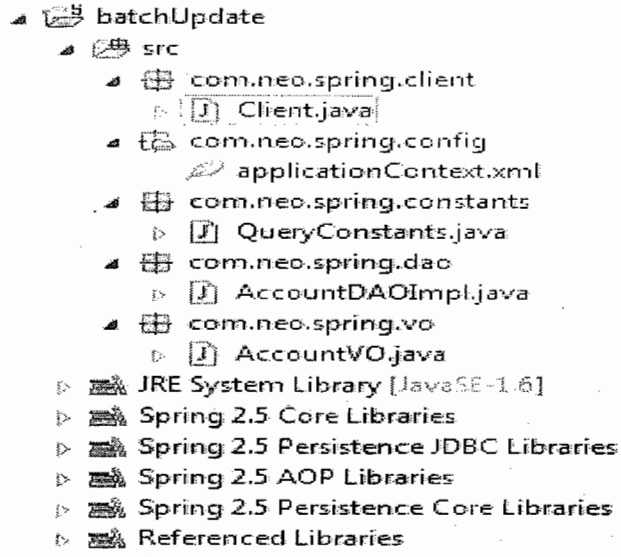
Before execution Account table

ACCNO	NAME	BAL
1004	somu	9800
1005	sekharreddy	2490
1003	somasekhar	7890

batchUpdate:

In some cases, you may required to perform multiple DML operations at a time. If you perform those operations individually it will cause performance issue. So instead of performing DML operations individually we perform them as a batch, so that the performance of the system will be increased.

In Spring JDBC framework, you can use JdbcTemplate class batchUpdate()to perform the batch operations.



applicationContext.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3.     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xmlns:p="http://www.springframework.org/schema/p"
5.     xsi:schemaLocation="http://www.springframework.org/schema/beans
6. http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
7.
8. <!--  Apache datasource configuration -->
9.     <bean id="apacheDatasourceRef"
10.                          class="org.apache.commons.dbcp.BasicDataSource">
11.      <property name="driverClassName"
12.                                  value="oracle.jdbc.driver.OracleDriver" />
13.      <property name="url" value="jdbc:oracle:thin:@localhost:1521:XE" />
14.      <property name="username" value="system" />
15.      <property name="password" value="tiger" />
16.    </bean>
17.
18.    <bean id="accountDAORef" class="com.neo.spring.dao.AccountDAOImpl">
19.          <property name="dataSource" ref=" apacheDatasourceRef " />
20.    </bean>
21.
22.    </beans>
```

AccountVO.java

```
1. package com.neo.spring.vo;
2. public class AccountVO {
3.     private int accno;
4.     private String name;
```

```
5.    private double balance;
6.
7.    public AccountVO(){
8.    }
9.
10.   public AccountVO(int accno, String name, double balance) {
11.         this.accno = accno;
12.         this.name = name;
13.         this.balance = balance;
14.   }
15.   //setters & getters
16.
17.   @Override
18.   public String toString() {
19.         return "ACCNO :"+accno+" || NAME :"+name+" ||BALANCE :"+balance;
20.         }
21.
22.   }
```

QueryConstants.java

```
1. package com.neo.spring.constants;
2.
3. public interface QueryConstants {
4.    public static String INSERT_QUERY_001 = "INSERT INTO ACCOUNT
5.                                            VALUES(1001,'somu', 4560)";
6.    public static String UPDATE_QUERY_002 = "UPDATE ACCOUNT SET
7.                                       NAME='yerragudi' WHERE ACCNO=1004";
8.    public static String DELETE_QUERY_003 = "DELETE FROM ACCOUNT WHERE
9.                                                    ACCNO=1003";
10.   public static String INSERT_QUERY_004 =  "INSERT INTO ACCOUNT
11.                                                  VALUES(?,?,?)";
12.   }
```

AccountDAOImpl.java

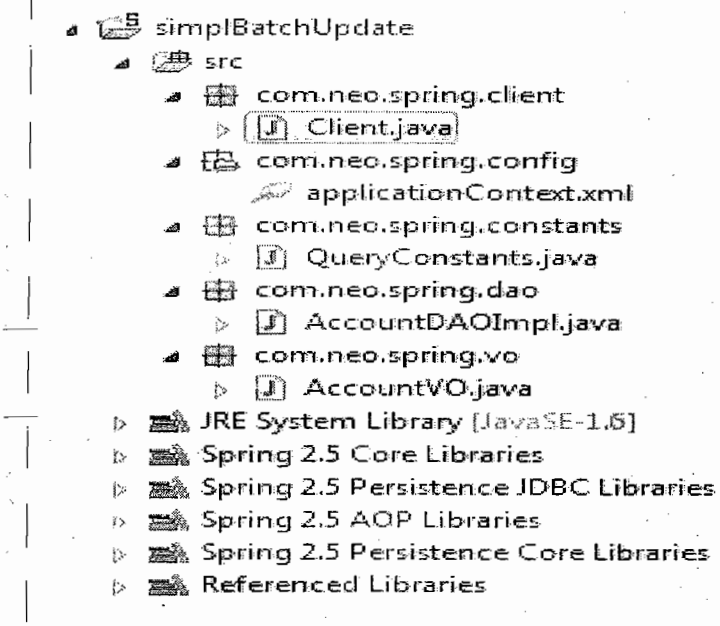
```
1. package com.neo.spring.dao;
2. import java.sql.PreparedStatement;
3. import java.sql.SQLException;
4. import java.util.List;
5. import org.springframework.jdbc.core.BatchPreparedStatementSetter;
6. import org.springframework.jdbc.core.support.JdbcDaoSupport;
7.
8. import com.neo.spring.constants.QueryConstants;
9. import com.neo.spring.vo.AccountVO;
10.
11.   public class AccountDAOImpl extends JdbcDaoSupport{
12.
13.         public int[] usingBatchUpdate001(){
14.               return getJdbcTemplate().batchUpdate(new String[]{
15.                                    QueryConstants.INSERT_QUERY_001,
                                      QueryConstants.UPDATE_QUERY_002,
16.                                   QueryConstants.DELETE_QUERY_003});
17.         }
18.
19.         public int[] usingBatchUpdate002(final List<AccountVO>
20.                                                   accountVOs){
```

```
21.             return getJdbcTemplate().batchUpdate(
22.                   QueryConstants.INSERT_QUERY_004,
23.                new BatchPreparedStatementSetter() {
24.
25.                @Override
26.                public void setValues(PreparedStatement ps, int count)
27.                                                 throws SQLException {
28.                         AccountVO vo= accountVOs.get(count);
29.                         ps.setInt(1, vo.getAccno());
30.                         ps.setString(2, vo.getName());
31.                         ps.setDouble(3, vo.getBalance());
32.
33.                }
34.
35.                @Override
36.                public int getBatchSize() {
37.                         return accountVOs.size();
38.                }
39.             });
40.       }
41.
42.  }
```

Client.java

```
1. package com.neo.spring.client;
2.
3. import java.util.ArrayList;
4. import java.util.List;
5.
6. import org.springframework.context.ApplicationContext;
7. import org.springframework.context.support.ClassPathXmlApplicationContext;
8.
9. import com.neo.spring.dao.AccountDAOImpl;
10.   import com.neo.spring.vo.AccountVO;
11.
12.   public class Client {
13.        private static ApplicationContext context = new
14.           ClassPathXmlApplicationContext(
15.                   "com/neo/spring/config/applicationContext.xml");
16.
17.        public static void main(String[] args) {
18.             AccountDAOImpl impl= (AccountDAOImpl)
19.                                  context.getBean("accountDAORef");
20.
21.             sop("---usingBatchUpdate001---");
22.             int ra[] = impl.usingBatchUpdate001();
23.             for(int i : ra)
24.                  sop(i);
25.
26.
27.             sop("---usingBatchUpdate002---");
28.             List<AccountVO> accountVOs = new ArrayList<AccountVO>();
29.             for(int i=2001; i<=2010; i++){
30.                  accountVOs.add(new AccountVO(i, "sekhar-"+i, i+100));
31.             }
```

```
32.                ra[] = impl.usingBatchUpdate002(accountVOs);
33.                for(int i : ra)
34.                      sop(i);
35.
36.          }
37.
38.          public static void sop(Object object) {
39.                System.out.println(object);
40.          }
41.
42.    }
```



applicationContext.xml

```
<!—SAME AS ABOVE-->
```

AccountVO.java

```
<!—SAME AS ABOVE-->
```

QueryConstants.java

```
1. package com.neo.spring.constants;
2.
3. public interface QueryConstants {
4.     public static String INSERT_QUERY_001 =  "INSERT INTO ACCOUNT
5.                                                VALUES(?,?,?)";
6.     public static String INSERT_QUERY_002 =  "INSERT INTO ACCOUNT
7.                                 VALUES(:accno, :name, :balance)";
8. }
```

AccountDAOImpl.java

```
1. package com.neo.spring.dao;
2.
3. import java.util.ArrayList;
4. import java.util.List;
5.
6. import org.springframework.jdbc.core.namedparam.BeanPropertySqlParameterSource;
```

```
7. import org.springframework.jdbc.core.namedparam.SqlParameterSource;
8. import org.springframework.jdbc.core.namedparam.SqlParameterSourceUtils;
9. import org.springframework.jdbc.core.simple.SimpleJdbcDaoSupport;
10.
11.   import com.neo.spring.constants.QueryConstants;
12.   import com.neo.spring.vo.AccountVO;
13.
14.   public class AccountDAOImpl extends SimpleJdbcDaoSupport{
15.
16.         public int[] usingBatchUpdate001(final List<AccountVO>
17.                                                      accountVOs){
18.
19.               List<Object[]> batchArgs = new ArrayList<Object[]>();
20.               for(AccountVO vo : accountVOs)
21.                     batchArgs.add(new Object[]{vo.getAccno(),
22.                                 vo.getName(), vo.getBalance()});
23.               return getSimpleJdbcTemplate().batchUpdate(
24.                         QueryConstants.INSERT_QUERY_001, batchArgs);
25.         }
26.         public int[] usingBatchUpdate002(final List<AccountVO>
27.                                                      accountVOs){
28.
29.               SqlParameterSource []batchArgs = new
30.                                   SqlParameterSource[accountVOs.size()];
31.               for(int i=0 ; i<batchArgs.length; i++){
32.                     batchArgs[i]= new BeanPropertySqlParameterSource(
33.                                                accountVOs.get(i));
34.               }
35.               return getSimpleJdbcTemplate().batchUpdate(
36.                       QueryConstants.INSERT_QUERY_002, batchArgs);
37.         }
38.
39.         public int[] usingBatchUpdate003(final List<AccountVO>
40.                                                      accountVOs){
41.
42.               SqlParameterSource []batchArgs =
43.                SqlParameterSourceUtils.createBatch(accountVOs.toArray());
44.               return getSimpleJdbcTemplate().batchUpdate(
45.                                 QueryConstants.INSERT_QUERY_002, batchArgs);
46.         }
47.
48.   }
```

Client.java

```
1. package com.neo.spring.client;
2.
3. import java.util.ArrayList;
4. import java.util.List;
5.
6. import org.springframework.context.ApplicationContext;
7. import org.springframework.context.support.ClassPathXmlApplicationContext;
8.
9. import com.neo.spring.dao.AccountDAOImpl;
10.   import com.neo.spring.vo.AccountVO;
11.
```

```
12.  public class Client {
13.        private static ApplicationContext context = new
14.             ClassPathXmlApplicationContext(
15.                   "com/neo/spring/config/applicationContext.xml");
16.
17.        public static void main(String[] args) {
18.              AccountDAOImpl impl= (AccountDAOImpl)
19.                                   context.getBean("accountDAORef");
20.
21.              sop("---usingBatchUpdate001---");
22.              List<AccountVO> accountVOs = new ArrayList<AccountVO>();
23.              for(int i=2001; i<=2005; i++){
24.                    accountVOs.add(new AccountVO(i, "sekhar-"+i, i+100));
25.              }
26.
27.              int ra[] = impl.usingBatchUpdate001(accountVOs);
28.              for(int i : ra)
29.                    sop(i);
30.
31.              sop("---usingBatchUpdate002---");
32.              accountVOs = new ArrayList<AccountVO>();
33.              for(int i=3001; i<=3005; i++){
34.                    accountVOs.add(new AccountVO(i, "sekhar-"+i, i+100));
35.              }
36.
37.              ra[] = impl.usingBatchUpdate002(accountVOs);
38.              for(int i : ra)
39.                    sop(i);
40.
41.
42.              sop("---usingBatchUpdate003---");
43.              accountVOs = new ArrayList<AccountVO>();
44.              for(int i=4001; i<=4005; i++){
45.                    accountVOs.add(new AccountVO(i, "sekhar-"+i, i+100));
46.              }
47.
48.              ra[] = impl.usingBatchUpdate003(accountVOs);
49.              for(int i : ra)
50.                    sop(i);
51.
52.
53.        }
54.
55.        public static void sop(Object object) {
56.              System.out.println(object);
57.        }
58.
59.  }
```

Before execution Account table

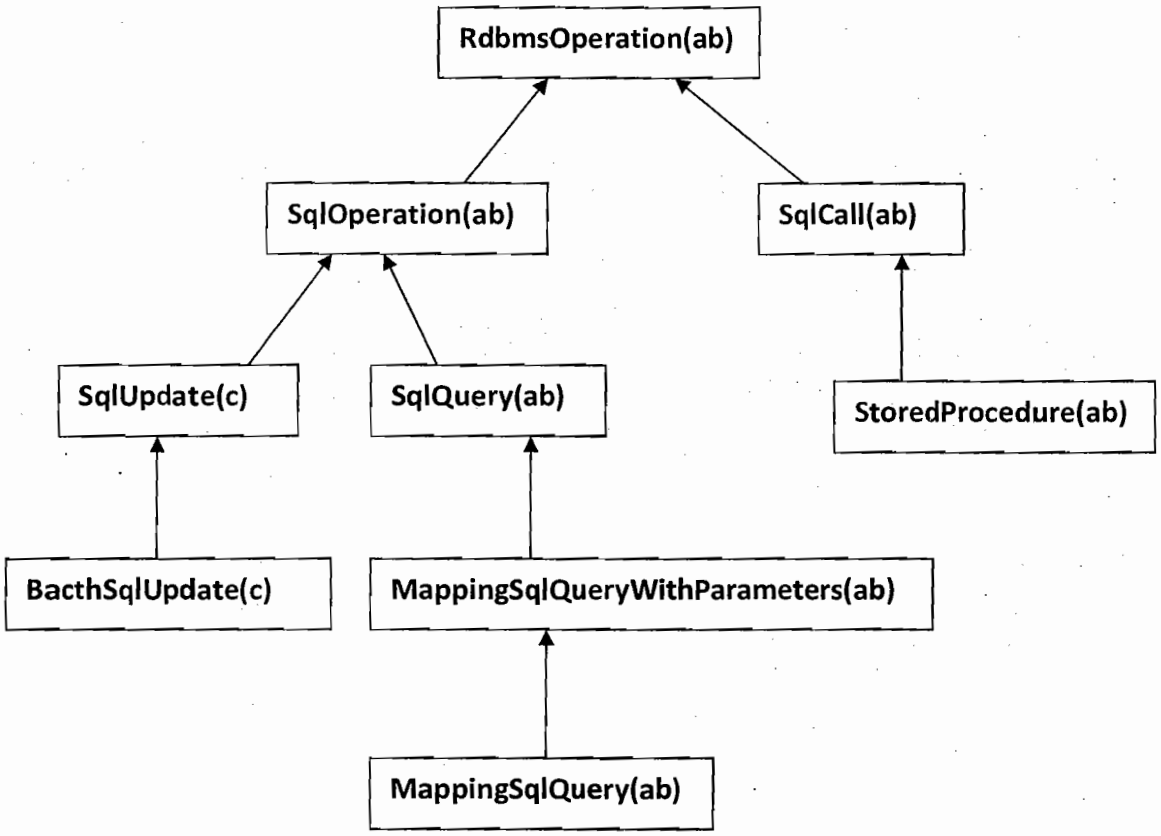
ACCNO	NAME	BAL
1004	somu	9800
1005	sekharreddy	2490
1003	somasekhar	7890

Modeling JDBC operations as Java objects:

The org.springframework.jdbc.object package contains classes that allow one to access the database in a mc object-oriented manner. By way of an example, one can execute queries and get the results back as a list containing business objects with the relational column data mapped to the properties of the business object. One can also execute stored procedures and run update, delete and insert statements.

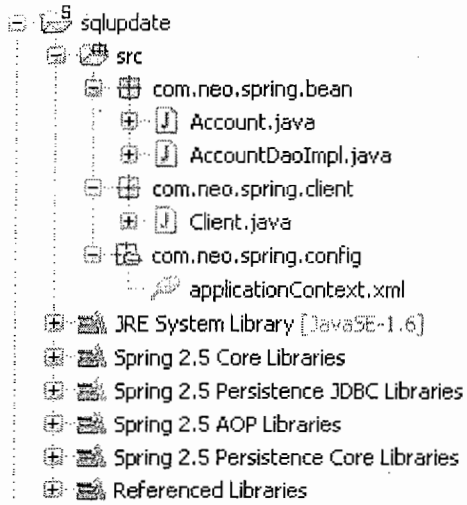
NOTE: There is a view borne from experience acquired in the field amongst some of the Spring developers that the various RDBMS operation classes described below (with the exception of the StoredProcedure class) can often be replaced with straight JdbcTemplate calls... often it is simpler to use and plain easier to read a DAO method that simp calls a method on a JdbcTemplate direct (as opposed to encapsulating a query as a full-blown class).

Hierarchy of fine-grained object classes



SqlUpdate

The SqlUpdate class encapsulates an SQL update. Like a query, an update object is reusable, and like all RdbmsOperation classes, an update can have parameters and is defined in SQL. This class provides a number of update(..) methods analogous to the execute(..) methods of query objects. This class is concrete. Although it can be subclassed it can easily be parameterized by setting SQL and declaring parameters.



applicationContext.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3.     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xmlns:p="http://www.springframework.org/schema/p"
5.     xsi:schemaLocation="http://www.springframework.org/schema/beans
6. http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
7.
8. <!--  Apache datasource configuration -->
9.     <bean id="apacheDatasourceRef"
10.                          class="org.apache.commons.dbcp.BasicDataSource">
11.      <property name="driverClassName"
12.                                  value="oracle.jdbc.driver.OracleDriver" />
13.      <property name="url" value="jdbc:oracle:thin:@localhost:1521:XE" />
14.      <property name="username" value="system" />
15.      <property name="password" value="tiger" />
16.    </bean>
17.
18.    <bean id="accountDAORef" class="com.neo.spring.dao.AccountDAOImpl">
19.          <property name="dataSource" ref=" apacheDatasourceRef " />
20.    </bean>
21.
22.    </beans>
```

Account.java

```
1. package com.neo.spring.bean;
2. public class Account {
3.     private int accno;
4.     private String name;
5.     private double bal;
6.
7.     // getters & setters
8. }
```

AccountDaoImpl.java

```
1. package com.neo.spring.bean;
2. import java.sql.Types;
3. import javax.sql.DataSource;
```

```
4. import org.springframework.jdbc.core.SqlParameter;
5. import org.springframework.jdbc.core.support.JdbcDaoSupport;
6. import org.springframework.jdbc.object.SqlUpdate;
7.
8. public class AccountDaoImpl extends JdbcDaoSupport {
9.
10.         public void usingSqlUpdate001(){
11.               DataSource dataSource= getDataSource();
12.               String query="INSERT INTO ACCOUNT VALUES
13.                                          (2001,'SEKHAR',4500)";
14.               AccountSqlUpdate accountSqlUpdate = new
15.                           AccountSqlUpdate(dataSource, query);
16.               int ra = accountSqlUpdate.update();
17.               System.out.println("No.of rows affected : "+ra);
18.         }
19.
20.         public void usingSqlUpdate002(){
21.               DataSource dataSource= getDataSource();
22.               String query="INSERT INTO ACCOUNT VALUES (?,?,?)";
23.               AccountSqlUpdate accountSqlUpdate = new
24.                                    AccountSqlUpdate(dataSource, query);
25.               int ra = accountSqlUpdate.update(new Object[]{2002,"somu",
26.                                                                  6790});
27.               System.out.println("No.of rows affected : "+ra);
28.         }
29.
30.         private class AccountSqlUpdate extends SqlUpdate{
31.               public AccountSqlUpdate(DataSource ds, String query) {
32.                     super(ds, query);
33.                     declareParameter(new SqlParameter("ACCNO",
34.                                                       Types.INTEGER));
35.                     declareParameter(new SqlParameter("NAME",
36.                                                     Types.VARCHAR));
37.                     declareParameter(new SqlParameter("BAL",
38.                                                       Types.DOUBLE));
39.                     compile();
40.               }
41.         }
42.   }
```

Client.java

```
1. package com.neo.spring.client;
2. import org.springframework.context.ApplicationContext;
3. import org.springframework.context.support.ClassPathXmlApplicationContext;
4. import com.neo.spring.bean.AccountDaoImpl;
5.
6. public class Client {
7.    private static ApplicationContext context = new
8.           ClassPathXmlApplicationContext(
9.               "com/neo/spring/config/applicationContext.xml");
10.
11.         public static void main(String[] args) {
12.               AccountDaoImpl dao = (AccountDaoImpl)
13.                           context.getBean("accountDaoImpl");
14.
```

```
15.                 dao.usingSqlUpdate001();
16.                 dao.usingSqlUpdate002();
17.                 System.out.println("success");
18.         }
19.     }
```

Output

Before executing the Account table is:

ACCNO	NAME	BAL
1001	sekhar	2900
1002	soma	3400

First execute dao.usingSqlUpdate001() method without giving any declareParameter(), because here we are hard coding the values. After executing dao.usingSqlUpdate001() the account table is:

ACCNO	NAME	BAL
1001	sekhar	2900
1002	soma	3400
2001	SEKHAR	4500

Now execute dao.usingSqlUpdate002() method make sure to give declareParameter().

After executed Account table is:

ACCNO	NAME	BAL
1001	sekhar	2900
1002	soma	3400
2001	SEKHAR	4500
2002	somu	6790

SqlQuery

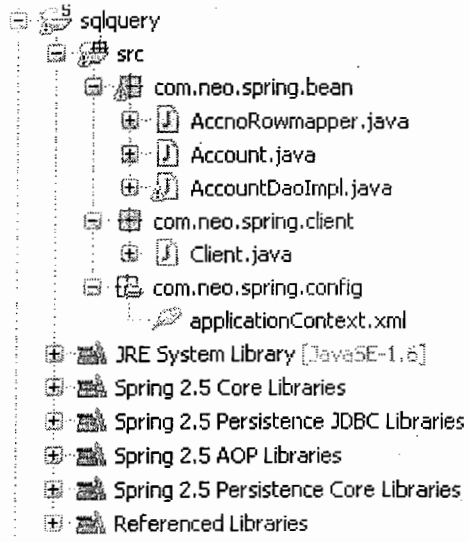
SqlQuery is a reusable, threadsafe class that encapsulates an SQL query. Subclasses must implement the newRowMapper(..) method to provide a RowMapper instance that can create one object per row obtained from iterating over the ResultSet that is created during the execution of the query. The SqlQuery class is rarely used directly since the MappingSqlQuery subclass provides a much more convenient implementation for mapping rows to Java classes. Other implementations that extend SqlQuery are MappingSqlQueryWithParameters and UpdatableSqlQuery.

MappingSqlQuery

MappingSqlQuery is a reusable query in which concrete subclasses must implement the abstract mapRow(..) method to convert each row of the supplied ResultSet into an object. Find below a brief example of a custom query that maps the data from the customer relation to an instance of the Customer class.

We provide a constructor for this customer query that takes the DataSource as the only parameter. In this constructor we call the constructor on the superclass with the DataSource and the SQL that should be executed to retrieve the rows for this query. This SQL will be used to create a PreparedStatement so it may contain place holders for any parameters to be passed in during execution. Each parameter must be declared using the declareParameter method passing in an SqlParameter. The SqlParameter takes a name and the JDBC type as defined in java.sql.Types. After all parameters have been defined we call the compile() method so the statement can be prepared and later be executed.

MappingSqlQuery



applicationContext.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3.     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xmlns:p="http://www.springframework.org/schema/p"
5.     xsi:schemaLocation="http://www.springframework.org/schema/beans
6. http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
7.
8. <!--  Apache datasource configuration -->
9.     <bean id="apacheDatasourceRef"
10.                          class="org.apache.commons.dbcp.BasicDataSource">
11.      <property name="driverClassName"
12.                                value="oracle.jdbc.driver.OracleDriver" />
13.      <property name="url" value="jdbc:oracle:thin:@localhost:1521:XE" />
14.      <property name="username" value="system" />
15.      <property name="password" value="tiger" />
16.    </bean>
17.
18.    <bean id="accountDAORef" class="com.neo.spring.dao.AccountDAOImpl">
19.          <property name="dataSource" ref=" apacheDatasourceRef " />
20.    </bean>
21.
22.    </beans>
```

Account.java

```
1. package com.neo.spring.bean;
2. public class Account {
3.     private int accno;
4.     private String name;
5.     private double bal;
6.
7.     // getters & setters
8. }
```

AccountDaoImpl.java

```
1. package com.neo.spring.bean;
2. import java.sql.ResultSet;
```

```
3. import java.sql.SQLException;
4. import java.sql.Types;
5. import java.util.Iterator;
6. import java.util.List;
7. import javax.sql.DataSource;
8. import org.springframework.jdbc.core.SqlParameter;
9. import org.springframework.jdbc.core.support.JdbcDaoSupport;
10.   import org.springframework.jdbc.object.MappingSqlQuery;
11.
12.   public class AccountDaoImpl extends JdbcDaoSupport {
13.
14.         public void usingSqlQuery001() {
15.               String query = "SELECT * FROM ACCOUNT WHERE ACCNO=1001";
16.               DataSource dataSource = getDataSource();
17.               AccountMappingSqlQuery mappingSqlQuery = new
18.                           AccountMappingSqlQuery(dataSource, query);
19.               List<Account> accounts = mappingSqlQuery.execute();
20.               Account account = accounts.get(0);
21.               System.out.println(account.getAccno());
22.               System.out.println(account.getName());
23.               System.out.println(account.getBal());
24.
25.         }
26.
27.         public void usingSqlQuery002() {
28.               String query = "SELECT * FROM ACCOUNT";
29.               DataSource dataSource = getDataSource();
30.               AccountMappingSqlQuery mappingSqlQuery = new
31.                           AccountMappingSqlQuery(dataSource, query);
32.               List<Account> accounts = mappingSqlQuery.execute();
33.               Iterator<Account> iterator = accounts.iterator();
34.               while (iterator.hasNext()) {
35.                     Account account = iterator.next();
36.                     System.out.println(account.getAccno());
37.                     System.out.println(account.getName());
38.                     System.out.println(account.getBal());
39.               }
40.         }
41.
42.         public void usingSqlQuery003() {
43.               String query = "SELECT * FROM ACCOUNT WHERE BAL>?";
44.               DataSource dataSource = getDataSource();
45.               AccountMappingSqlQuery mappingSqlQuery = new
46.                           AccountMappingSqlQuery(dataSource, query);
47.               List<Account> accounts = mappingSqlQuery.execute(new
48.                                                      Object[]{100});
49.               Iterator<Account> iterator = accounts.iterator();
50.               while (iterator.hasNext()) {
51.                     Account account = iterator.next();
52.                     System.out.println(account.getAccno());
53.                     System.out.println(account.getName());
54.                     System.out.println(account.getBal());
55.               }
56.
57.         }
```

```
58.
59.         private class AccountMappingSqlQuery extends MappingSqlQuery {
60.
61.               public AccountMappingSqlQuery(DataSource dataSource, String
62.                                                                     query){
63.                     super(dataSource, query);
64.                     declareParameter(new SqlParameter("BAL",
65.                                                       Types.DOUBLE));
66.                     compile();
67.               }
68.
69.               protected Object mapRow(ResultSet rs, int rowNum) throws
70.                                                       SQLException {
71.
72.                     Account account = new Account();
73.                     account.setAccno(rs.getInt(1));
74.                     account.setName(rs.getString(2));
75.                     account.setBal(rs.getDouble(3));
76.
77.                     return account;
78.               }
79.         }
80.   }
```

Client.java

```
1. package com.neo.spring.client;
2. import org.springframework.context.ApplicationContext;
3. import org.springframework.context.support.ClassPathXmlApplicationContext;
4. import com.neo.spring.bean.AccountDaoImpl;
5.
6. public class Client {
7.    private static ApplicationContext context = new
8.            ClassPathXmlApplicationContext(
9.                "com/neo/spring/config/applicationContext.xml");
10.
11.        public static void main(String[] args) {
12.              AccountDaoImpl dao = (AccountDaoImpl)
13.                                    context.getBean("accountDaoImpl");
14.
15.              dao.usingSqlQuery001();
16.   //         dao.usingSqlQuery002();
17.   //         dao.usingSqlQuery003();
18.              System.out.println("success");
19.        }
20.   }
```

Account Table:

ACCNO	NAME	BAL
1001	sekhar	2900
1002	soma	3400
2001	SEKHAR	4500
2002	somu	6790

While executing dao.usingSqlQuery001(); there is no need of declareParameter() because we are hard coding the value, so remove declareParameter() otherwise we will get InvalidDataAccessApiUsageException: 0 parameters were supplied, but 1 in parameters were declared in class

Output for dao.usingSqlQuery001():

```
1001
sekhar
2900.0
Success
```

While executing dao.usingSqlQuery002(); also there is no need of declareParameter() because we are not giving place holder, so make sure to remove declareParameter() otherwise we will get exception.

Output for dao.usingSqlQuery002():

```
1001
sekhar
2900.0
1002
soma
3400.0
2001
SEKHAR
4500.0
2002
somu
6790.0
Success
```

While executing dao.usingSqlQuery003(); here we have place holder so we have to give declareParameter().

Output for dao.usingSqlQuery003():

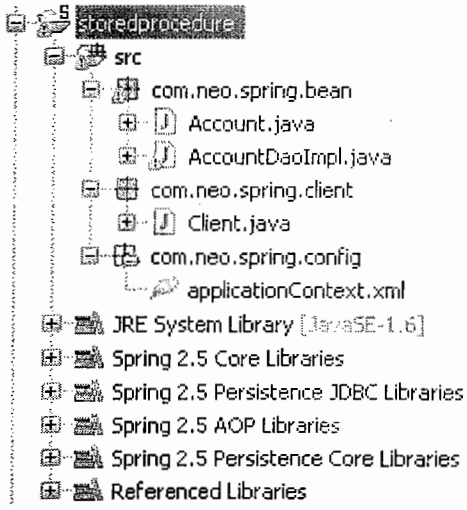
```
1001
sekhar
2900.0
1002
soma
3400.0
2001
SEKHAR
4500.0
2002
somu
6790.0
Success
```

StoredProcedure

The StoredProcedure class is a superclass for object abstractions of RDBMS stored procedures. This class is abstract, and its various execute(..) methods have protected access, preventing use other than through a subclass that offers tighter typing.

To use the stored procedure functionality one has to create a class that extends StoredProcedure. There are no input parameters, but there is an output parameter that is declared as a date type using the class SqlOutParameter. The execute() method returns a map with an entry for each declared output parameter using the parameter name as the key.

If one needs to pass parameters to a stored procedure (that is the stored procedure has been declared as having one or more input parameters in its definition in the RDBMS), one would code a strongly typed execute(..) method which would delegate to the superclass' (untyped) execute(Map parameters).



applicationContext.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3.     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xmlns:p="http://www.springframework.org/schema/p"
5.     xsi:schemaLocation="http://www.springframework.org/schema/beans
6. http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
7.
8. <!--  Apache datasource configuration -->
9.     <bean id="apacheDatasourceRef"
10.                          class="org.apache.commons.dbcp.BasicDataSource">
11.      <property name="driverClassName"
12.                                  value="oracle.jdbc.driver.OracleDriver" />
13.      <property name="url" value="jdbc:oracle:thin:@localhost:1521:XE" />
14.      <property name="username" value="system" />
15.      <property name="password" value="tiger" />
16.    </bean>
17.
18.    <bean id="accountDAORef" class="com.neo.spring.dao.AccountDAOImpl">
19.          <property name="dataSource" ref=" apacheDatasourceRef " />
20.    </bean>
21.
22.    </beans>
```

Account.java

```
1. package com.neo.spring.bean;
2. public class Account {
3.     private int accno;
4.     private String name;
5.     private double bal;
6.
7.     // getters & setters
8. }
```

AccountDaoImpl.java

```
1. package com.neo.spring.bean;
2. import java.sql.Types;
3. import java.util.HashMap;
```

```
4. import java.util.Map;
5. import javax.sql.DataSource;
6. import org.springframework.jdbc.core.SqlOutParameter;
7. import org.springframework.jdbc.core.SqlParameter;
8. import org.springframework.jdbc.core.support.JdbcDaoSupport;
9. import org.springframework.jdbc.object.StoredProcedure;
10.
11.   public class AccountDaoImpl extends JdbcDaoSupport {
12.
13.         public void usingStoredProcedure() {
14.               Map<String, Object> params = new HashMap<String, Object>();
15.               params.put("accno_p", 6004);
16.               params.put("name_p", "somu");
17.               params.put("bal_p", 5800);
18.
19.               String procedureName = "account_insert_procedure";
20.               DataSource dataSource = getDataSource();
21.               AccountStoredProcedure procedure = new
22.                        AccountStoredProcedure(dataSource, procedureName);
23.               Map resultMap = procedure.execute(params);
24.               System.out.println(resultMap);
25.         }
26.
27.         private class AccountStoredProcedure extends StoredProcedure {
28.               public AccountStoredProcedure(DataSource dataSource, String
29.                                                          procedureName) {
30.                     super(dataSource, procedureName);
31.                     declareParameter(new SqlParameter("accno_p",
32.                                                          Types.INTEGER));
33.                     declareParameter(new SqlParameter("name_p",
34.                                                          Types.VARCHAR));
35.                     declareParameter(new SqlParameter("bal_p",
36.                                                           Types.DOUBLE));
37.                     declareParameter(new SqlOutParameter("result",
38.                                                          Types.VARCHAR));
39.                     compile();
40.               }
41.         }
42.   }
```

Client.java

```
1. package com.neo.spring.client;
2. import org.springframework.context.ApplicationContext;
3. import org.springframework.context.support.ClassPathXmlApplicationContext;
4. import com.neo.spring.bean.AccountDaoImpl;
5.
6. public class Client {
7.    private static ApplicationContext context = new
8.            ClassPathXmlApplicationContext(
9.               "com/neo/spring/config/applicationContext.xml");
10.
11.        public static void main(String[] args) {
12.              AccountDaoImpl dao = (AccountDaoImpl)
13.                                 context.getBean("accountDaoImpl");
14.              dao.usingStoredProcedure();
```

```
15.                 System.out.println("success");
16.         }
17.   }
```

Procedure

```
CREATE OR REPLACE PROCEDURE SCOTT.ACCOUNT_INSERT_PROCEDURE (accno_p number, name_p
varchar2, bal_p number, result_p out varchar2) IS

BEGIN

   insert into account values(accno_p, name_p, bal_p);
   commit;
 result_p:='success';

END ACCOUNT_INSERT_PROC;
/
```

Output

{result=success}
Success

Account Table:

Before execution

ACCNO	NAME	BAL
1001	sekhar	2900
1002	soma	3400
2001	SEKHAR	4500
2002	somu	6790

AfterExecution

ACCNO	NAME	BAL
6004	somu	5800
1001	sekhar	2900
1002	soma	3400
2001	SEKHAR	4500
2002	somu	6790