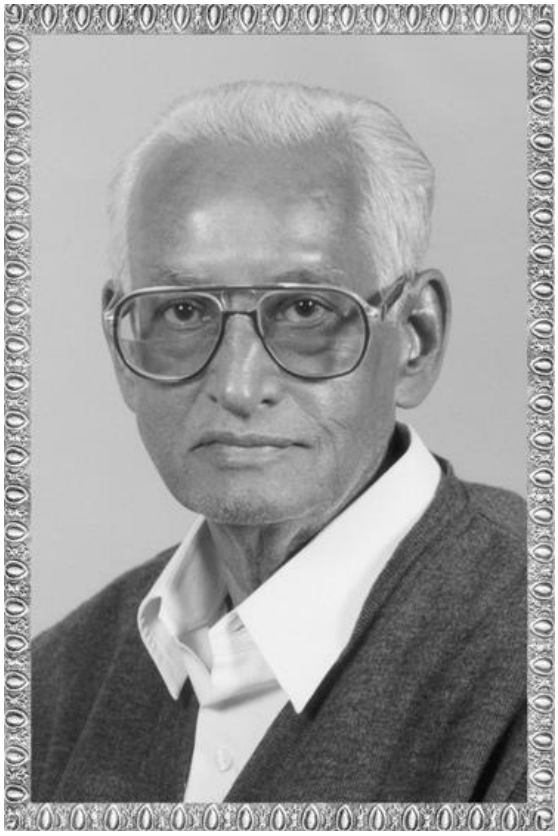
SHREE H. N. SHUKLA GROUP OF COLLEGES

(Affiliated to Saurashtra University & Gujarat Technological University)



Lt. Shree Chimanbhai Shukla

M.Sc. I.T. SEM-1 - Advanced Java

Shree H.N.Shukla College Campus,
Street No. 2, Vaishali Nagar,
Nr. Amrapali Railway Crossing,
Raiya Road, Rajkot.
Ph. (0281)2440478, 2472590

Shree H.N.Shukla College
Street No. 3, Vaishali Nagar,
Nr. Amrapali Railway Crossing,
Raiya Road, Rajkot.
Ph. (0281)2471645

Website: www.hnshukla.com
Email : hnsinfo@hnshukla.com

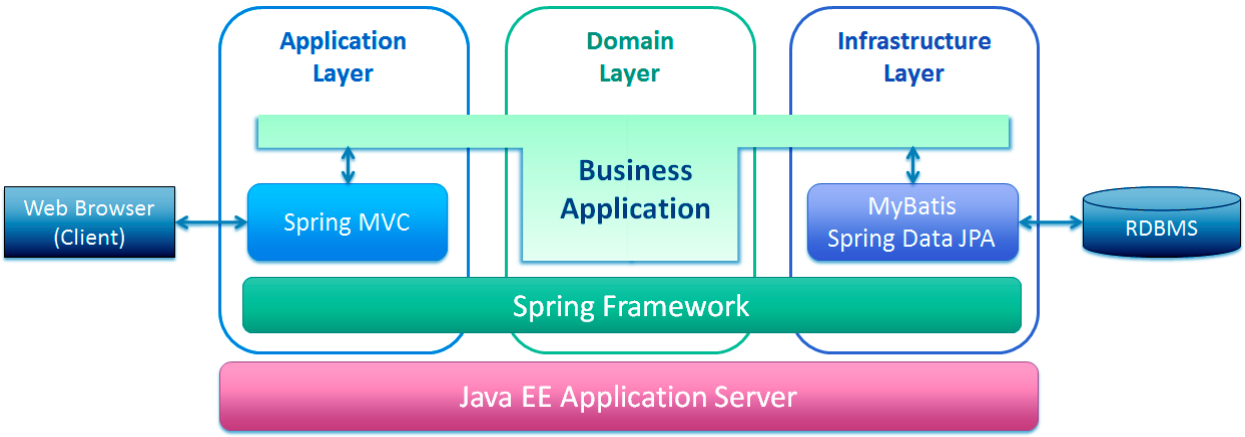
2 – Vaishalinagar
Nr. Amrapali Railway Crossing
Raiya Road
Rajkot – 360001
Ph.No–(0281)2440478,2472590

3 – Vaishalinagar
Nr. Amrapali Railway Crossing
Raiya Road
Rajkot - 360001
Ph.No–(0281)2471645

Unit - 1

What is Spring?

- Spring is one of the most popular **Java-based** application frameworks.
- The Spring Framework was developed by **Rod Johnson in 2003**.
- The Spring Framework is **an open-source** framework that can be used to develop Java applications with ease and at a rapid pace.
- It is **a lightweight framework** that also provides well-defined infrastructure support for developing applications in Java. In other words, you can say that Spring handles the infrastructure so that you can focus more on developing your application.
- Spring is modular in nature, which means that you can use the parts that you need instead of using the whole framework.
- Using the Spring Framework, you can build Java applications as well as other kinds of web applications.



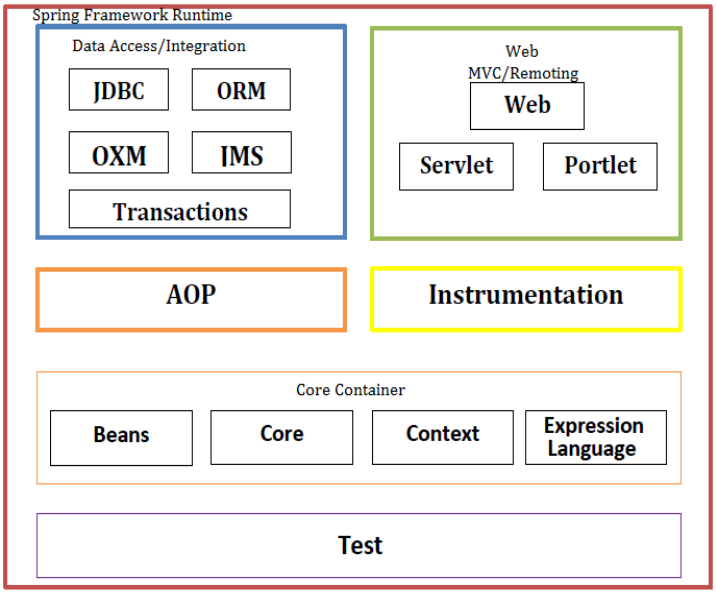
Sr no	Question	Answer
1	Who was developed the Spring Framework?	**Rod Johnson**
2	When Spring Framework was developed?	**2003**
3	True/False Spring is Lightweight Framework.	**True**
4	True/False Spring is an open source framework which is used to develop Java applications	**True**

2 – Vaishalinagar
Nr. Amrapali Railway Crossing
Raiya Road
Rajkot – 360001
Ph.No–(0281)2440478,2472590

3 – Vaishalinagar
Nr. Amrapali Railway Crossing
Raiya Road
Rajkot - 360001
Ph.No–(0281)2471645

Explain Spring Modules.

- **Spring** is a modular framework and comprises various independent modules who play around the core framework capabilities.
- The modular characteristic of the **Spring** framework allows to pick up the selected modules which are appropriate to particular use case instead of pulling all modules in the code.
- This brings great flexibility and ease in development to adapt what you want to be part of the system.
- Spring provides around 20 modules which can be grouped logically in the following categories:



1. Core container

The core of Spring framework comprises to following modules:

1.1 Spring core:

- It's the heart of the Spring framework, mainly provides IoC container implementation and dependency management capabilities.
- The IoC container isolates the configuration of beans and their dependency information from application code.

1.2 Spring beans

- In the Spring world, the application objects are called beans.

- This module provides factory pattern implementation called bean factory, which is used to create and maintain the lifecycle of application objects (beans).

1.3 Spring context

- This module is built on top of the solid foundation provided by Spring core and Spring beans modules.
- It is responsible for loading configuration (XML or Annotation) and then use Spring core and Spring beans modules to provide access to the application objects (beans).

1.4 Spring Expression Language (SpEL)

- As an extension of Expression Language (EL) of JSP 2.1 specification, Spring Expression Language (shortly Spring EL or just SpEL) is used to access and manipulates application objects (beans) directly in JSP on the fly.

Advantages:

- **Cleaner code:** Spring EL allows writing cleaner code by avoiding scriptlets in the JSP and thus increase code readability.
- **Property configuration:** Allows setting of getters and setters of the bean with ease.
- **Access application object (bean):** It allows to access any bean defined in spring's application context. Also, Spring EL can be used to access any method of the controller directly from JSP.
- **Misc:** Spring EL allows various interacting operation like selection, projection aggregation, etc. It also allows various operator to performs arithmetical operations.

2. Data access

Spring modules fall under data access category are mainly used for data manipulation. It comprises to following modules:

2.1 Java Data-Base Connectivity(JDBC)

- This module is responsible for providing a low-level code to deal with JDBC abstraction.
- It is used to interact with a database with standard JDBC API.

2.2 Object Relationship Mapping (ORM)

- Instead of providing its own ORM, Spring supports integration with various ORM frameworks like Hibernate, JPA, JDO, etc with this module.

- This brings consistency and portability to application code regardless of various data access technologies.

2.3 Transaction

- Spring manages the database transactions of type programmatic and declarative with this module.
- For this purpose, this module closely works with ORM and JDBC modules.
- All enterprise level transaction can be managed by this module.

3. Web

Spring provides a set of modules that are used mainly to build web-based applications as follows:

3.1 Web

- As its name suggests, this modules facilitates basic web-related integration features like a multipart file upload.
- In background, it uses a custom tag in JSP for this.
- It also initializes the Spring IoC container and makes it ready in the web application context.

3.2 Servlet

- This module contains Model-View-Controller implementation for Spring MVC in a web-based application.
- Obviously, the benefits of MVC is a clear separation of view layer from the model and controller who control the flow between them.
- This module provides various custom tags and validations which can be used in JSP.

3.3 Portlet

- The Servlet module provides MVC implementation for a servlet-based web application.
- Similarly, this module provides MVC implementation for portlet based application.
- It mirrors the functionalities of a servlet-based web module in a portlet context.

3.4 Web-socket

- This module is responsible for providing two-way communication between client and server with WebSocket implementation in a web application.

3.5 Reactive

- Spring provides reactive-stack web application support with this module.
- It's also known as WebFlux which was added since Spring 5.0 to support reactive programming.

4. Testing

This module is responsible for providing backing support of the unit as well as integration testing of Spring components with well-known testing frameworks like JUnit, TestNg, etc. Additionally, It helps in loading the Spring application context in a consistent manner. Moreover, it assists in providing mock objects which are used to test the application code in an isolated environment.

5. Integration

Spring integration support lightweight messaging mechanism to interact with an external system with the declarative approach. It provides a high-level abstraction for supporting remote integration.

Spring framework provides a set of APIs to deal with various technology by implementing remote services with POJO based model.

Currently, Spring support integration of the following technologies:

- **JAX-WS:** Spring provides out of the box remote support for web service implemented through JAX-WS.
- **JMS:** Spring provides a set of classes with underlying protocol to interact remote system with JMS.
- **AMQP:** Spring provides an implementation of AMQP protocol and allows to interact with high-level abstraction.
- **RMI:** Spring allows exposing custom services through RMI support.
- **Spring's HTTP invoke:** Spring supports the invocation of remote service through native java serialization which is accessible over HTTP protocol.

6. Miscellaneous

Spring provides few miscellaneous features which may help to make the application development with ease and make them more stable to get a steady output as follows:

- **AOP:** Spring provides Aspect Oriented Programming (AOP) programming model which is deeply helpful to implement various interceptors. This substantially decouples the common functionalities and allows to embed them in a plug and play fashion.
- **Aspects:** This module is useful to write the code with AspectJ – a feature rich, powerful and mature AOP framework.
- **Instrumentation:** Spring provides class loader implementation through this module which is required for certain application servers.

Sr no	Question	Answer
1	Which module is theheart of the Spring framework?	**Spring Core**
2	Application objects are called _____?	**Beans**
3	ORM stands for?	**Object Relation Mapping**
4	Which programming model is deeply helpful to implement various interceptors in spring.	**AOP**

List the application of spring.

- The spring framework latest version provides sound support to a multitude of application architectures including
  - web application
  - messaging application
  - transaction data architecture.
  - Servlet-based Spring MVC web application.
  - Spring WebFlux reactive web application.
  - Enterprise java applications

Example of Spring in Myeclipse

- Creating spring application in myeclipse IDE is simple.
- You don't need to be worried about the jar files required for spring application because myeclipse IDE takes care of it.

- Let's see the simple steps to create the spring application in myeclipse IDE.
    - **Create the java project**
    - **Add spring capabilities**
    - **Create the class**
    - **Create the xml file to provide the values**
    - **Create the test class**

Let's see the 5 steps to create the first spring application using myeclipse IDE.

1) Create the Java Project

- Go to **File menu - New - project - Java Project**.
- Write the project name e.g. **firstspring** - Finish.
- Now the java project is created.

2) Add spring capabilities:

- Go to **Myeclipse** menu - **Project Capabilities - Add spring capabilities - Finish**.
- Now the spring jar files will be added.
- For the simple application we need only core library i.e. selected **by default.**

3) Create Java class

- Create **Student class** having **name** property
- The name of the student will be provided by the xml file.
- This is simple bean class, containing only one property **name** with its **getters and setters method**.
- This class contains one extra method named displayInfo() that prints the student name by the hello message.
- To create the java class, **Right click on src - New - class** - Write the class name e.g. **Student** - finish. Write the following code:

```
1. public class Student {
2. private String name;
3. public String getName() {
4.     return name;  }
5.  public void setName(String name) {
```

```
6.  this.name = name;  }
7.  public void displayInfo(){
8.     System.out.println("Hello: "+name);  }
9.  }
```

4) Create the xml file

- In case of myeclipse IDE, you don't need to create the xml file as myeclipse does this for yourselves.
- The bean element is used to define **the bean** for the given class.
- The property sub element of bean specifies the **property of the Student class** named name.
- The **value** specified in the property element will be set in the Student class object by the IOC container.
- Open the applicationContext.xml file, and write the following code:

```
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-3.0.xsd">
8.   <bean id="studentbean" class=" Student">
9.  <property name="name" value="Ms. Bijal Parekh"></property>
10. </bean>  </beans>
```

5) Create the test class

- Create the java class e.g. Test.
- Here we are getting the object of Student class from the IOC container using the getBean( ) method of BeanFactory.
- Let's see the code of test class.

```
1. import org.springframework.beans.factory.BeanFactory;
2. import org.springframework.beans.factory.xml.XmlBeanFactory;
3. import org.springframework.core.io.ClassPathResource;
4. import org.springframework.core.io.Resource;
5.   public class Test {
6. public static void main(String[] args) {
7.     Resource resource=new ClassPathResource("applicationContext.xml");
8.     BeanFactory factory=new XmlBeanFactory(resource);
9.     Student student=(Student)factory.getBean("studentbean");
10.    student.displayInfo();
11. }  }
```

- The Resource object represents the information of applicationContext.xml file.
- The Resource is the interface and the ClassPathResource is the implementation class of the Reource interface.
- The BeanFactory is responsible to return the bean. The XmlBeanFactory is the implementation class of the BeanFactory.
- There are many methods in the BeanFactory interface.
- One method is getBean(), which returns the object of the associated class.
- Now run the Test class. You will get the **output Hello: Ms. Bijal Parekh**.

Example of Spring in Eclipse

- Here, we are going to create a simple application of spring framework using eclipse IDE.
- Let's see the simple steps to create the spring application in Eclipse IDE.
- The only difference is that we are suppose to add jar file manually and create xml files manually.
    - **Create the java project**
    - **Add spring jar files**
    - **Create the class**
    - **Create the xml file to provide the values**
    - **Create the test class**

Let's see the 5 steps to create the first spring application using eclipse IDE.

1) Create the Java Project

- Go to **File** menu - **New** - **project** - **Java Project**.
- Write the project name e.g. firstspring - **Finish**.
- Now the java project is created.

2) Add spring jar files

- There are mainly three jar files required to run this application.
  - **org.springframework.core-3.0.1.RELEASE-A**
  - **com.springsource.org.apache.commons.logging-1.1.1**
  - **org.springframework.beans-3.0.1.RELEASE-A**
- To run this example, you need to load only spring core jar files.
- To load the jar files in eclipse IDE, **Right click on your project** - **Build Path** - **Add external archives** - **select all the required jar files** - **finish.**.

3) Create Java class

- Create Student class as above.

4) Create the xml file

- To create the xml file click on **src** - **new** - **file** - give the file name such as applicationContext.xml - finish.
- Open the applicationContext.xml file, and write the code as given above.

5) Create the test class

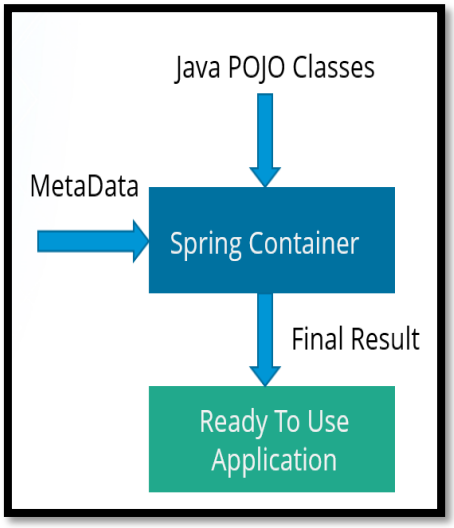
- Create Test class as above.
- Output will be same as given.

IoC Container

Questions

- Explain IoC Container
- Explain BeanFactory
- Explain ApplicationContext
- Difference between BeanFactory and ApplicationContext

- The IoC container is responsible to instantiate, configure and assemble the objects. The IoC container gets in formations from the XML file and works accordingly.
- Spring IoC stands for **Inversion of Control**.
- It is the heart of the Spring Framework. The important tasks performed by the IoC container are:
  1. **Instantiating the bean**
  2. **Wiring the beans together**
  3. **Configuring the beans**
  4. **Managing the bean's entire life-cycle**



There are two types of IoC containers. They are:

1. **BeanFactory**
2. **ApplicationContext**

1. BeanFactory

- It is an interface defined in **org.springframework.beans.factory.BeanFactory**.
- Bean Factory provides the basic support for Dependency Injection.
- It is based on factory design pattern which creates the beans of any type.
- BeanFactory follows **lazy-initialization** technique which means beans are loaded as soon as bean factory instance is created but the beans are created only when getBean() method is called.
- The XmlBeanFactory is the implementation class for the BeanFactory interface.
- To use the BeanFactory, you need to create the instance of XmlBeanFactory class as shown below:

```
BeanFactory beanFactory = new XmlBeanFactory(new
                                ClassPathResource("beans.xml"))
```

2. ApplicationContext

- It is an interface defined in **org.springframework.context.ApplicationContext**.
- It is the **advanced Spring container** and is built on top of the BeanFactory interface.
- ApplicationContext supports the features supported by Bean Factory but also provides some additional functionalities.

- ApplicationContext follows **eager-initialization technique** which means instance of beans are created as soon as you create the instance of Application context.
- The ClassPathXmlApplicationContext class is the implementation class of ApplicationContext interface.
- You need to instantiate the ClassPathXmlApplicationContext class to use the ApplicationContext as shown below:

```
ApplicationContext context=new
                    ClassPathXmlApplicationContext("beans.xml");
```

Sr no	Question	Answer
1	Which container is responsible to instantiate, configure and assemble the objects in spring.	**IOC**
2	Types of IOC container	**BeanFactory and ApplicationContext**
3	BeanFactory follows _____ technique.	**Lazy-initialization**
4	ApplicationContext follows _____ technique	**Eagar intialization**

Difference between ApplicationContext and the BeanFactory

ApplicationContext	BeanFactory
ApplicationContext--Support Annotation based dependency Injection.-@Autowired, @PreDestroy	BeanFactory-Does not support the Annotation based dependency Injection.
ApplicationContext- Application contexts can publish events to beans that are registered as listeners	BeanFactory-Does not Support
ApplicationContext-Support internationalization (I18N) messages.	BeanFactory-Does not support way to access Message Bundle(internationalization (I18N)
ApplicationContext-Support many enterprise services such JNDI access, EJB integration, remoting.	BeanFactory-Doesn't support.
ApplicationContext-- its By default support Aggresive loading.	BeanFactory-By default its support Lazy loading

Dependency Injection in Spring

Questions:

- Explain Dependency Injection in Spring.
- What is Dependency Lookup.

- Dependency Injection (DI) is a design pattern that removes the dependency from the programming code so that it can be easy to manage and test the application.
- Dependency Injection makes our programming code loosely coupled.
- To understand the DI better, Let's understand the Dependency Lookup (DL) first:

Dependency Lookup

- The Dependency Lookup is an approach where we get the resource after demand. There can be various ways to get the resource for example:

```
A obj = new AImpl();
```

- In such way, we get the resource(instance of A class) directly by new keyword. Another way is factory method:

```
A obj = A.getA();
```

- This way, we get the resource (instance of A class) by calling the static factory method getA().

Alternatively, we can get the resource by JNDI (Java Naming Directory Interface) as:

1. Context ctx = new InitialContext();
2. Context environmentCtx = (Context) ctx.lookup("java:comp/env");
3. A obj = (A)environmentCtx.lookup("A");

There can be various ways to get the resource to obtain the resource. Let's see the problem in this approach.

Problems of Dependency Lookup

There are mainly two problems of dependency lookup.

- **tight coupling** The dependency lookup approach makes the code tightly coupled. If resource is changed, we need to perform a lot of modification in the code.

2 – Vaishalinagar
Nr. Amrapali Railway Crossing
Raiya Road
Rajkot – 360001
Ph.No–(0281)2440478,2472590

3 – Vaishalinagar
Nr. Amrapali Railway Crossing
Raiya Road
Rajkot - 360001
Ph.No–(0281)2471645

- **Not easy for testing** This approach creates a lot of problems while testing the application especially in black box testing.

Dependency Injection

- The Dependency Injection is a design pattern that removes the dependency of the programs.
- In such case we provide the information from the external source such as XML file.
- It makes our code loosely coupled and easier for testing. In such case we write the code as:

```
1. class Employee{
2. Address address;
3. Employee(Address address){
4. this.address=address;
5. }
6. public void setAddress(Address address){
7. this.address=address;
8. }
9. }
```

- In such case, instance of Address class is provided by external source such as XML file either by constructor or setter method.
- Two ways to perform Dependency Injection in Spring framework:
    - **By Constructor Injection:** It is accomplished when the container invokes a class constructor with a number of arguments where each representing a dependency on the other class
    - **By Setter Injection :** It is accomplished by the container calling the setter methods on the beans after invoking a no-argument constructor or a no-argument static factory method to instantiate the bean.

Advantages of Dependency Injection (DI)

- Reducing the dependency to each other objects in an application
- Every object in an application could be individually unit tested with different mock implementations
- Loosely coupled and promotes decoupling of an application

- Promotes reusability of the code or objects in the different applications
- Promotes a logical abstraction of the components

Sr no	Question	Answer
1	What is a design pattern that removes the dependency from the programming code so that it can be easy to manage and test the application.	**Dependency Injection**
2	What makes our programming code loosely coupled.	**Dependency Injection**
3	Two ways to perform Dependency Injection are?	**Constructor Injection and Setter injection**

Dependency Injection by Constructor

We can inject the dependency by constructor. The **<constructor-arg>** subelement of **<bean>** is used for constructor injection. Here we are going to inject

1. primitive and String-based values
2. Dependent object (contained object)
3. Collection values etc.

Injecting primitive based values

Let's see the simple example to inject primitive and string-based values. We have created three files here:

Example 1:

- Employee.java
- applicationContext.xml
- Test.java

Employee.java

It is a simple class containing two fields id and name. There are four constructors and one method in this class.

```
1. public class Employee
2. {
3. private int id;
4. private String name;
```

```
5.  public Employee() {System.out.println("def cons");}
6.  public Employee(int id) {this.id = id;}
7.  public Employee(String name) {  this.name = name;}
8.  public Employee(int id, String name) {
9.     this.id = id;
10.    this.name = name;
11. }
12. public void show()
13. {      System.out.println(id+" "+name);  }
14. }
```

applicationContext.xml

- We are providing the information into the bean by this file.
- The **constructor-arg** element invokes the constructor.
- In such case, parameterized constructor of **int type** will be invoked.
- The **value attribute of constructor-arg element** will assign the specified value.
- The type attribute specifies that **int parameter** constructor will be invoked.

```
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-3.0.xsd">
8.  <bean id="emp" class=" Employee">
9.  <constructor-arg value="10" type="int"></constructor-arg>
10. </bean>
11. </beans>
```

Test.java

- This class gets the bean from the applicationContext.xml file and calls the show method.

```
1. import org.springframework.beans.factory.BeanFactory;
2. import org.springframework.beans.factory.xml.XmlBeanFactory;
3. import org.springframework.core.io.*;
4. public class Test {
5.     public static void main(String[] args) {
6.         Resource r=new ClassPathResource("applicationContext.xml");
7.         BeanFactory factory=new XmlBeanFactory(r);
8.         Employee s=(Employee)factory.getBean("emp");
9.         s.show();
10.   } }
```

Output:10 null

Injecting string-based values

- If you don't specify the type attribute in the constructor-arg element, by default string type constructor will be invoked.

```
1. ....
2. <bean id="emp" class="Employee">
3. <constructor-arg value="10"></constructor-arg>
4. </bean>
5. ....
```

- If you change the bean element as given above, string parameter constructor will be invoked and the output will be 0 10.

Output:0 10

- You may also pass the string literal as following:

```
1. <bean id="e" class="com.javatpoint.Employee">
2. <constructor-arg value="Gujarat"></constructor-arg>
3. </bean>
```

Output:0 Gujarat

- You may pass integer literal and string both as following

```
1. <bean id="e" class="com.javatpoint.Employee">
2. <constructor-arg value="10" type="int" ></constructor-arg>
3. <constructor-arg value="Gujarat"></constructor-arg>
4. </bean>
```

Output:10 Gujarat

Constructor Injection with Dependent Object

- If there is HAS-A relationship between the classes, we create the instance of dependent object first then pass it as an argument of the main class constructor.
- Here, our scenario is **Employee HAS-A Address**.
- The Address class object will be termed as the dependent object.
- Let's see the Address class first:

Address.java

This class contains three properties, one constructor and toString() method to return the values of the object.

```
1. public class Address {
2. private String city;
3. private String state;
4. private String country;
5. public Address(String city, String state, String country) {
6.     super();
7.     this.city = city;
8.     this.state = state;
9.     this.country = country;  }
10. public String toString(){
11.     return city+" "+state+" "+country;
12. }
13. }
```

Employee.java

2 – Vaishalinagar
Nr. Amrapali Railway Crossing
Raiya Road
Rajkot – 360001
Ph.No–(0281)2440478,2472590

3 – Vaishalinagar
Nr. Amrapali Railway Crossing
Raiya Road
Rajkot - 360001
Ph.No–(0281)2471645

It contains three properties id, name and address(dependent object) ,two constructors and show() method to show the records of the current object including the depedent object.

```
1. public class Employee {
2. private int id;
3. private String name;
4. private Address address;//Aggregation
5. public Employee() {System.out.println("def cons");}
6. public Employee(int id, String name, Address address) {
7.     super();
8.     this.id = id;
9.     this.name = name;
10.    this.address = address;  }
11. void show(){
12.    System.out.println(id+" "+name);
13.    System.out.println(address.toString());
14. }  }
```

applicationContext.xml

The **ref** attribute is used to define the reference of another object, such way we are passing the dependent object as an constructor argument.

```
1.  <bean id="a1" class=" Address">
2. <constructor-arg value="Rajkot"></constructor-arg>
3. <constructor-arg value="GUJARAT"></constructor-arg>
4. <constructor-arg value="India"></constructor-arg>
5. </bean>
6. <bean id="e" class=" Employee">
7. <constructor-arg value="12" type="int"></constructor-arg>
8. <constructor-arg value="Bijal Parekh"></constructor-arg>
9. <constructor-arg>
10. <ref bean="a1"/>
11. </constructor-arg>  </bean>
```

Test.java

This class gets the bean from the applicationContext.xml file and calls the show method.

```
1. import org.springframework.beans.factory.BeanFactory;
2. import org.springframework.beans.factory.xml.XmlBeanFactory;
3. import org.springframework.core.io.*;
4. public class Test {
5.     public static void main(String[] args) {
6.         Resource r=new ClassPathResource("applicationContext.xml");
7.         BeanFactory factory=new XmlBeanFactory(r);
8.         Employee s=(Employee)factory.getBean("e");
9.         s.show();
10.    } }
```

Constructor Injection with Collection

- We can inject collection values by constructor in spring framework. There can be used three elements inside the **constructor-arg** element.
- It can be:
  1. **List (String based and non-string)**
  2. **Set (String based and non-string)**
  3. **Map (String based and non-string)**
- Each collection can have string based and non-string based values.

Constructor Injection with List Collection Example (String Based)

- In this example, we are taking the example of Forum where **One question can have multiple answers**. There are three pages:
  1. **Question.java**
  2. **applicationContext.xml**
  3. **Test.java**
- In this example, we are using list that can have duplicate elements, you may use set that have only unique elements.

- But, you need to change list to set in the applicationContext.xml file and List to Set in the Question.java file.

Question.java

- This class contains three properties, two constructors and displayInfo() method that prints the information.
- Here, we are using **List** to contain the multiple answers.

```
1. import java.util.Iterator;
2. import java.util.List;
3. public class Question {
4. private int id;
5. private String name;
6. private List<String> answers;
7. public Question() {}
8. public Question(int id, String name, List<String> answers) {
9.     super();
10.    this.id = id;
11.    this.name = name;
12.    this.answers = answers;
13. }
14. public void displayInfo(){
15.    System.out.println(id+" "+name);
16.    System.out.println("answers are:");
17.    Iterator<String> itr=answers.iterator();
18.    while(itr.hasNext()){
19.        System.out.println(itr.next());
20.    }
21. }
22. }
```

2 – Vaishalinagar
Nr. Amrapali Railway Crossing
Raiya Road
Rajkot – 360001
Ph.No–(0281)2440478,2472590

3 – Vaishalinagar
Nr. Amrapali Railway Crossing
Raiya Road
Rajkot - 360001
Ph.No–(0281)2471645

applicationContext.xml

The list element of **constructor-arg** is used here to define the list.

```
1. <bean id="q" class=" Question">
2. <constructor-arg value="111"></constructor-arg>
3. <constructor-arg value="What is java?"></constructor-arg>
4. <constructor-arg>
5. <list>
6. <value>Java is a programming language</value>
7. <value>Java is a Platform</value>
8. <value>Java is an Island of Indonasia</value>
9. </list>
10. </constructor-arg>
11. </bean>
12. </beans>
```

Test.java

This class gets the bean from the applicationContext.xml file and calls the displayInfo method.

```
1. import org.springframework.beans.factory.BeanFactory;
2. import org.springframework.beans.factory.xml.XmlBeanFactory;
3. import org.springframework.core.io.ClassPathResource;
4. import org.springframework.core.io.Resource;
5. public class Test {
6. public static void main(String[] args) {
7.     Resource r=new ClassPathResource("applicationContext.xml");
8.     BeanFactory factory=new XmlBeanFactory(r);
9.     Question q=(Question)factory.getBean("q");
10.    q.displayInfo();
11. }
12. }
```

2 – Vaishalinagar
Nr. Amrapali Railway Crossing
Raiya Road
Rajkot – 360001
Ph.No–(0281)2440478,2472590

3 – Vaishalinagar
Nr. Amrapali Railway Crossing
Raiya Road
Rajkot - 360001
Ph.No–(0281)2471645

C.I. with List Collection Example (non-String Based)

- If we have dependent object in the collection, we can inject these information by using the **ref** element inside the **list**, **set** or **map**.
- In this example, we are taking the example of Forum where **One question can have multiple answers**.
- But Answer has its own information such as answerId, answer and postedBy.
  1. **Question.java**
  2. **Answer.java**
  3. **applicationContext.xml**
  4. **Test.java**

Question.java

This class contains three properties, two constructors and displayInfo() method that prints the information. Here, we are using List to contain the multiple answers.

```
1. import java.util.*;
2. public class Question {
3. private int id;
4. private String name;
5. private List<Answer> answers;
6. public Question() {}
7. public Question(int id, String name, List<Answer> answers) {
8.     super();
9.     this.id = id;
10.    this.name = name;
11.    this.answers = answers;
12.    }
13. public void displayInfo(){
14.    System.out.println(id+" "+name);
15.    System.out.println("answers are:");
16.    Iterator<Answer> itr=answers.iterator();
17.    while(itr.hasNext()){
18.        System.out.println(itr.next());
19. } } }
```

2 – Vaishalinagar
Nr. Amrapali Railway Crossing
Raiya Road
Rajkot – 360001
Ph.No–(0281)2440478,2472590

3 – Vaishalinagar
Nr. Amrapali Railway Crossing
Raiya Road
Rajkot - 360001
Ph.No–(0281)2471645

Answer.java

This class has three properties id, name and by with constructor and toString() method.

```
1. public class Answer {
2. private int id;
3. private String name;
4. private String by;
5. public Answer() {}
6. public Answer(int id, String name, String by) {
7.     super();
8.     this.id = id;
9.     this.name = name;
10.    this.by = by;
11. }
12. public String toString(){
13.    return id+" "+name+" "+by;  }
14. }
```

applicationContext.xml

The **ref** element is used to define the reference of another bean. Here, we are using **bean** attribute of **ref** element to specify the reference of another bean.

```
1. <bean id="ans1" class="Answer">
2. <constructor-arg value="1"></constructor-arg>
3. <constructor-arg value="Java is a programming language"></constructor-arg>
4. <constructor-arg value="John"></constructor-arg>
5. </bean>
6. <bean id="ans2" class=" Answer">
7. <constructor-arg value="2"></constructor-arg>
8. <constructor-arg value="Java is a Platform"></constructor-arg>
9. <constructor-arg value="Ravi"></constructor-arg>
10. </bean>
```

```
11. <bean id="q" class=" Question">
12. <constructor-arg value="111"></constructor-arg>
13. <constructor-arg value="What is java?"></constructor-arg>
14. <constructor-arg>
15. <list>
16. <ref bean="ans1"/>
17. <ref bean="ans2"/>
18. </list>
19. </constructor-arg>
20. </bean>
21. </beans>
```

Test.java

This class gets the bean from the applicationContext.xml file and calls the displayInfo method.

```
1. import org.springframework.beans.factory.BeanFactory;
2. import org.springframework.beans.factory.xml.XmlBeanFactory;
3. import org.springframework.core.io.ClassPathResource;
4. import org.springframework.core.io.Resource;
5. public class Test {
6. public static void main(String[] args) {
7.     Resource r=new ClassPathResource("applicationContext.xml");
8.     BeanFactory factory=new XmlBeanFactory(r);
9.     Question q=(Question)factory.getBean("q");
10.    q.displayInfo();
11. } }
```

C. I. with Map Collection Example (String Based)

- In this example, we are using **map** as the answer that have answer with posted username. Here, we are using key and value pair both as a string.
- Like previous examples, it is the example of forum where **one question can have multiple answers**.

Question.java

This class contains three properties, two constructors and displayInfo() method to display the information.

```
1. import java.util.*;
2. public class Question {
3. private int id;
4. private String name;
5. private Map<String,String> answers;
6. public Question() {}
7. public Question(int id, String name, Map<String, String> answers) {
8.     super();
9.     this.id = id;
10.    this.name = name;
11.    this.answers = answers;  }
12.    public void displayInfo(){
13.    System.out.println("question id:"+id);
14.    System.out.println("question name:"+name);
15.    System.out.println("Answers....");
16.    Set<Entry<String, String>> set=answers.entrySet();
17.    Iterator<Entry<String, String>> itr=set.iterator();
18.    while(itr.hasNext()){
19.        Entry<String,String> entry=itr.next();
20.        System.out.println("Answer:"+entry.getKey()+" Posted By:"+entry.getValue());      }
    }
21. }
```

applicationContext.xml

The **entry** attribute of **map** is used to define the key and value information.

```
1. <bean id="q" class="com.javatpoint.Question">
2. <constructor-arg value="11"></constructor-arg>
3. <constructor-arg value="What is Java?"></constructor-arg>
```

```
4. <constructor-arg>
5. <map>
6. <entry key="Java is a Programming Language"  value="Ajay Kumar"></entry>
7. <entry key="Java is a Platform" value="John Smith"></entry>
8. <entry key="Java is an Island" value="Raj Kumar"></entry>
9. </map>
10. </constructor-arg>
11. </bean>
```

Test.java

This class gets the bean from the applicationContext.xml file and calls the displayInfo() method.

```
1. import org.springframework.beans.factory.BeanFactory;
2. import org.springframework.beans.factory.xml.XmlBeanFactory;
3. import org.springframework.core.io.ClassPathResource;
4. import org.springframework.core.io.Resource;
5. public class Test {
6. public static void main(String[] args) {
7.     Resource r=new ClassPathResource("applicationContext.xml");
8.     BeanFactory factory=new XmlBeanFactory(r);
9.     Question q=(Question)factory.getBean("q");
10.    q.displayInfo();
11. }  }
```

C. I. with Map Collection Example (non-String Based)

- In this example, we are using **map** as the answer that have Answer and User.
- Here, we are using key and value pair both as an object. Answer has its own information such as answerId, answer and postedDate, User has its own information such as userId, username, emailId.
- Like previous examples, it is the example of forum where **one question can have multiple answers**.

2 – Vaishalinagar
Nr. Amrapali Railway Crossing
Raiya Road
Rajkot – 360001
Ph.No–(0281)2440478,2472590

3 – Vaishalinagar
Nr. Amrapali Railway Crossing
Raiya Road
Rajkot - 360001
Ph.No–(0281)2471645

Question.java

This class contains three properties, two constructors and displayInfo() method to display the information.

```
1. import java.util.*;
2. public class Question {
3. private int id;
4. private String name;
5. private Map<Answer,User> answers;
6. public Question() {}
7. public Question(int id, String name, Map<Answer, User> answers) {
8.     super();
9.     this.id = id;
10.    this.name = name;
11.    this.answers = answers;  }
12. public void displayInfo(){
13.    System.out.println("question id:"+id);
14.    System.out.println("question name:"+name);
15.    System.out.println("Answers....");
16.    Set<Entry<Answer, User>> set=answers.entrySet();
17.    Iterator<Entry<Answer, User>> itr=set.iterator();
18.    while(itr.hasNext()){
19.        Entry<Answer, User> entry=itr.next();
20.        Answer ans=entry.getKey();
21.        User user=entry.getValue();
22.        System.out.println(ans);
23.        System.out.println("Posted By:");
24.        System.out.println(user);
25.    }  }  }
```

Answer.java

```
1. import java.util.Date;
2. public class Answer {
3. private int id;
4. private String answer;
5. private Date postedDate;
6. public Answer() {}
7. public Answer(int id, String answer, Date postedDate) {
8.     super();
9.     this.id = id;
10.    this.answer = answer;
11.    this.postedDate = postedDate;  }
12. public String toString(){
13.    return "Id:"+id+" Answer:"+answer+" Posted Date:"+postedDate;  }  }
```

User.java

```
1. public class User {
2. private int id;
3. private String name,email;
4. public User() {}
5. public User(int id, String name, String email) {
6.     super();
7.     this.id = id;
8.     this.name = name;
9.     this.email = email;  }
10. public String toString(){
11.    return "Id:"+id+" Name:"+name+" Email Id:"+email;  }  }
```

2 – Vaishalinagar
Nr. Amrapali Railway Crossing
Raiya Road
Rajkot – 360001
Ph.No–(0281)2440478,2472590

3 – Vaishalinagar
Nr. Amrapali Railway Crossing
Raiya Road
Rajkot - 360001
Ph.No–(0281)2471645

applicationContext.xml

The **key-ref** and **value-ref** attributes of entry **element** is used to define the reference of bean in the map.

```
1.  <bean id="answer1" class=" Answer">
2.  <constructor-arg value="1"></constructor-arg>
3.  <constructor-arg value="Java is a Programming Language"></constructor-arg>
4.  <constructor-arg value="12/12/2001"></constructor-arg>
5.  </bean>
6.  <bean id="answer2" class=" Answer">
7.  <constructor-arg value="2"></constructor-arg>
8.  <constructor-arg value="Java is a Platform"></constructor-arg>
9.  <constructor-arg value="12/12/2003"></constructor-arg>
10. </bean>
11.   <bean id="user1" class=" User">
12. <constructor-arg value="1"></constructor-arg>
13. <constructor-arg value="Arun Kumar"></constructor-arg>
14. <constructor-arg value="arun@gmail.com"></constructor-arg>
15. </bean>
16. <bean id="user2" class=" User">
17. <constructor-arg value="2"></constructor-arg>
18. <constructor-arg value="Varun Kumar"></constructor-arg>
19. <constructor-arg value="Varun@gmail.com"></constructor-arg>
20. </bean>
21.   <bean id="q" class=" Question">
22. <constructor-arg value="1"></constructor-arg>
23. <constructor-arg value="What is Java?"></constructor-arg>
24. <constructor-arg>
25. <map>
26. <entry key-ref="answer1" value-ref="user1"></entry>
27. <entry key-ref="answer2" value-ref="user2"></entry>
28. </map>  </constructor-arg>  </bean>
```

Test.java

This class gets the bean from the applicationContext.xml file and calls the displayInfo() method to display the information.

```
1. import org.springframework.beans.factory.BeanFactory;
2. import org.springframework.beans.factory.xml.XmlBeanFactory;
3. import org.springframework.core.io*;
4. public class Test {
5. public static void main(String[] args) {
6.     Resource r=new ClassPathResource("applicationContext.xml");
7.     BeanFactory factory=new XmlBeanFactory(r);
8.     Question q=(Question)factory.getBean("q");
9.     q.displayInfo();
10. } }
```

C. I. Inheriting Bean in Spring

- By using the **parent** attribute of **bean**, we can specify the inheritance relation between the beans. In such case, parent bean values will be inherited to the current bean.
- Let's see the simple example to inherit the bean.

Employee.java

This class contains three properties, three constructor and show() method to display the values.

```
1. public class Employee {
2. private int id;
3. private String name;
4. private Address address;
5. public Employee() {}
6. public Employee(int id, String name) {
7.     super();
8.     this.id = id;
9.     this.name = name;  }
10. public Employee(int id, String name, Address address) {
```

```
11.     super();
12.     this.id = id;
13.     this.name = name;
14.     this.address = address;
15. }
16. void show(){
17.     System.out.println(id+" "+name);
18.     System.out.println(address);
19. }  }
```

Address.java

```
1.  public class Address {
2.  private String addressLine1,city,state,country;
3.  public Address(String addressLine1, String city, String state, String country) {
4.      super();
5.      this.addressLine1 = addressLine1;
6.      this.city = city;
7.      this.state = state;
8.      this.country = country;  }
9.  public String toString(){
10.     return addressLine1+" "+city+" "+state+" "+country;  } }
```

applicationContext.xml

```
1.  <bean id="e1" class="Employee">
2.  <constructor-arg value="101"></constructor-arg>
3.  <constructor-arg  value="Sachin"></constructor-arg>
4.  </bean>
5.    <bean id="address1" class="Address">
6.  <constructor-arg value="1 Shantivan park"></constructor-arg>
7.  <constructor-arg value="Rajkot"></constructor-arg>
8.  <constructor-arg value="Gujarat"></constructor-arg>
```

```
9.  <constructor-arg value="India"></constructor-arg>
10. </bean>
11. <bean id="e2" class="Employee" parent="e1">
12. <constructor-arg ref="address1"></constructor-arg>
13. </bean>
```

Test.java

This class gets the bean from the applicationContext.xml file and calls the show method.

```
1.  import org.springframework.beans.factory.BeanFactory;
2.  import org.springframework.beans.factory.xml.XmlBeanFactory;
3.  import org.springframework.core.io.ClassPathResource;
4.  import org.springframework.core.io.Resource;
5.  public class Test {
6.  public static void main(String[] args) {
7.      Resource r=new ClassPathResource("applicationContext.xml");
8.      BeanFactory factory=new XmlBeanFactory(r);
9.      Employee e1=(Employee)factory.getBean("e2");
10.     e1.show();
11. } }
```

Dependency Injection by Setter

- We can inject the dependency by setter method also.
- The **<property>** subelement of **<bean>** is used for setter injection.
- Here we are going to inject
  1. **primitive and String-based values**
  2. **Dependent object (contained object)**
  3. **Collection values etc.**

Injecting primitive based values

Let's see the simple example to inject primitive and string-based values by setter method. We have created three files here:

2 – Vaishalinagar
Nr. Amrapali Railway Crossing
Raiya Road
Rajkot – 360001
Ph.No–(0281)2440478,2472590

3 – Vaishalinagar
Nr. Amrapali Railway Crossing
Raiya Road
Rajkot - 360001
Ph.No–(0281)2471645

- Employee.java
- applicationContext.xml
- Test.java

Employee.java

It is a simple class containing three fields id, name and city with its setters and getters and a method to display these informations.

```
1. public class Employee {
2. private int id;
3. private String name;
4. private String city;
5. public int getId() {  return id;  }
6. public void setId(int id) {   this.id = id;  }
7. public String getName() {   return name;  }
8. public void setName(String name) {  this.name = name;  }
9. public String getCity() {   return city;  }
10. public void setCity(String city) {   this.city = city;  }
11. void display(){
12.     System.out.println(id+" "+name+" "+city); }  }
```

applicationContext.xml

We are providing the information into the bean by this file. The property element invokes the setter method. The value subelement of property will assign the specified value.

```
1. <bean id="obj" class=" Employee">
2. <property name="id">
3. <value>20</value>
4. </property>
5. <property name="name">
6. <value>Arun</value>
7. </property>
8. <property name="city">  <value>ghaziabad</value>  </property>
9.   </bean>
```

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

Test.java

This class gets the bean from the applicationContext.xml file and calls the display method.

```
1.  import org.springframework.beans.factory.BeanFactory;
2.  import org.springframework.beans.factory.xml.XmlBeanFactory;
3.  import org.springframework.core.io.*;
4.  public class Test {
5.      public static void main(String[] args) {
6.          Resource r=new ClassPathResource("applicationContext.xml");
7.          BeanFactory factory=new XmlBeanFactory(r);
8.          Employee e=(Employee)factory.getBean("obj");
9.          s.display();
10.     } }
```

Injecting String based values

- Like Constructor Injection, we can inject the dependency of another bean using setters.
- In such case, we use **property** element. Here, our scenario is **Employee HAS-A Address**.
- The Address class object will be termed as the dependent object.
- Let's see the Address class first:

Address.java

This class contains four properties, setters and getters and toString() method.

```
1.  public class Address {
2.  private String addressLine1,city,state,country;
3.  //getters and setters
4.  public String toString(){
5.      return addressLine1+" "+city+" "+state+" "+country;  }
```

Employee.java

It contains three properties id, name and address(dependent object) , setters and getters with displayInfo() method.

2 – Vaishalinagar
Nr. Amrapali Railway Crossing
Raiya Road
Rajkot – 360001
Ph.No–(0281)2440478,2472590

3 – Vaishalinagar
Nr. Amrapali Railway Crossing
Raiya Road
Rajkot - 360001
Ph.No–(0281)2471645

```
1. public class Employee {
2. private int id;
3. private String name;
4. private Address address;
5. //setters and getters
6. void displayInfo(){
7.     System.out.println(id+" "+name);
8.     System.out.println(address);  }  }
```

applicationContext.xml

The **ref** attribute of **property** elements is used to define the reference of another bean.

```
1. <bean id="address1" class="com.javatpoint.Address">
2. <property name="addressLine1" value="51,Lohianagar"></property>
3. <property name="city" value="Ghaziabad"></property>
4. <property name="state" value="UP"></property>
5. <property name="country" value="India"></property>
6. </bean>
7. <bean id="obj" class="com.javatpoint.Employee">
8. <property name="id" value="1"></property>
9. <property name="name" value="Sachin Yadav"></property>
10. <property name="address" ref="address1"></property>
11. </bean>
```

Test.java

This class gets the bean from the applicationContext.xml file and calls the displayInfo() method.

```
1. import org.springframework.beans.factory.BeanFactory;
2. import org.springframework.beans.factory.xml.XmlBeanFactory;
3. import org.springframework.context.ApplicationContext;
4. import org.springframework.context.support.ClassPathXmlApplicationContext;
5. import org.springframework.core.io.*;
6. public class Test {
```

```
7.  public static void main(String[] args) {
8.      Resource r=new ClassPathResource("applicationContext.xml");
9.      BeanFactory factory=new XmlBeanFactory(r);
10.     Employee e=(Employee)factory.getBean("obj");
11.     e.displayInfo();  }  }
```

Setter Injection with Collection

- We can inject collection values by setter method in spring framework. There can be used three elements inside the **property** element.
- It can be:
  1. **list**
  2. **set**
  3. **map**
- Each collection can have string based and non-string based values.

S.I. with List Collection Example (String Based)

- In this example, we are taking the example of Forum where **One question can have multiple answers**. There are three pages:
  1. **Question.java**
  2. **applicationContext.xml**
  3. **Test.java**

Question.java

This class contains three properties with setters and getters and displayInfo() method that prints the information. Here, we are using List to contain the multiple answers.

```
1.  import java.util.Iterator;
2.  import java.util.List;
3.  public class Question {
4.  private int id;
5.  private String name;
6.  private List<String> answers;
7.  //setters and getters
8.  public void displayInfo(){
9.      System.out.println(id+" "+name);
```

```
10.  System.out.println("answers are:");
11.  Iterator<String> itr=answers.iterator();
12.  while(itr.hasNext()){
13.      System.out.println(itr.next());
14.  } } }
```

applicationContext.xml

The list element of constructor-arg is used here to define the list.

```
1.  <bean id="q" class="com.javatpoint.Question">
2.  <property name="id" value="1"></property>
3.  <property name="name" value="What is Java?"></property>
4.  <property name="answers">
5.  <list>
6.  <value>Java is a programming language</value>
7.  <value>Java is a platform</value>
8.  <value>Java is an Island</value>
9.  </list>
10. </property>
11. </bean>
12. </beans>
```

Test.java

This class gets the bean from the applicationContext.xml file and calls the displayInfo method.

```
1.  import org.springframework.beans.factory.BeanFactory;
2.  import org.springframework.beans.factory.xml.XmlBeanFactory;
3.  import org.springframework.core.io.ClassPathResource;
4.  import org.springframework.core.io.Resource;
5.  public class Test {
6.  public static void main(String[] args) {
7.      Resource r=new ClassPathResource("applicationContext.xml");
8.      BeanFactory factory=new XmlBeanFactory(r);
9.      Question q=(Question)factory.getBean("q");
```

```
10.    q.displayInfo();
11. } }
```

S.I. with List Collection Example (non-String Based)

- If we have dependent object in the collection, we can inject these information by using the **ref** element inside the **list**, **set** or **map**. Here, we will use list, set or map element inside the **property** element.
- In this example, we are taking the example of Forum where **One question can have multiple answers**. But Answer has its own information such as answerId, answer and postedBy. There are four pages used in this example:
  1. **Question.java**
  2. **Answer.java**
  3. **applicationContext.xml**
  4. **Test.java**

Question.java

This class contains three properties, two constructors and displayInfo() method that prints the information. Here, we are using List to contain the multiple answers.

```
1.  import java.util.*;
2.  public class Question {
3.  private int id;
4.  private String name;
5.  private List<Answer> answers;
6.  //setters and getters
7.  public void displayInfo(){
8.      System.out.println(id+" "+name);
9.      System.out.println("answers are:");
10.     Iterator<Answer> itr=answers.iterator();
11.     while(itr.hasNext()){
12.         System.out.println(itr.next());
13.     } } }
```

Answer.java

This class has three properties id, name and by with constructor and toString() method.

```
1.  public class Answer {
```

```
2. private int id;
3. private String name;
4. private String by;
5. //setters and getters
6. public String toString(){   return id+" "+name+" "+by;  }  }
```

applicationContext.xml

The **ref** element is used to define the reference of another bean. Here, we are using **bean** attribute of **ref** element to specify the reference of another bean.

```
1. <bean id="answer1" class=" Answer">
2. <property name="id" value="1"></property>
3. <property name="name" value="Java is a programming language"></property>
4. <property name="by" value="Ravi Malik"></property>
5. </bean>
6. <bean id="answer2" class=" Answer">
7. <property name="id" value="2"></property>
8. <property name="name" value="Java is a platform"></property>
9. <property name="by" value="Sachin"></property>
10. </bean>
11.  <bean id="q" class="Question">
12. <property name="id" value="1"></property>
13. <property name="name" value="What is Java?"></property>
14. <property name="answers">
15. <list>  <ref bean="answer1"/>  <ref bean="answer2"/>  </list>
16. </property>
17. </bean>
```

Test.java

This class gets the bean from the applicationContext.xml file and calls the displayInfo method.

```
1. import org.springframework.beans.factory.BeanFactory;
2. import org.springframework.beans.factory.xml.XmlBeanFactory;
3. import org.springframework.core.io.ClassPathResource;
4. import org.springframework.core.io.Resource;
5. public class Test {
6. public static void main(String[] args) {
7.     Resource r=new ClassPathResource("applicationContext.xml");
8.     BeanFactory factory=new XmlBeanFactory(r);
9.     Question q=(Question)factory.getBean("q");
10.    q.displayInfo();
11. } }
```

Setter Injection with Map Example

In this example, we are using **map** as the answer for a question that have answer as the key and username as the value. Here, we are using key and value pair both as a string.

Like previous examples, it is the example of forum where **one question can have multiple answers**.

Question.java

This class contains three properties, getters & setters and displayInfo() method to display the information.

```
1. import java.util.*;
2. public class Question {
3. private int id;
4. private String name;
5. private Map<String,String> answers;
6. //getters and setters
7. public void displayInfo(){
8.     System.out.println("question id:"+id);
```

```
9.    System.out.println("question name:"+name);
10.   System.out.println("Answers....");
11.   Set<Entry<String, String>> set=answers.entrySet();
12.   Iterator<Entry<String, String>> itr=set.iterator();
13.   while(itr.hasNext()){
14.      Entry<String,String> entry=itr.next();
15.      System.out.println("Answer:"+entry.getKey()+" Posted By:"+entry.getValue());
16.   } } }
```

applicationContext.xml

The **entry** attribute of **map** is used to define the key and value information.

```
1.  <bean id="q" class=" Question">
2.  <property name="id" value="1"></property>
3.  <property name="name" value="What is Java?"></property>
4.  <property name="answers">
5.  <map>
6.  <entry key="Java is a programming language"  value="Bijal Parekh"></entry>
7.  <entry key="Java is a Platform" value="Mr Aditya G"></entry>
8.  </map>  </property>
9.  </bean>
```

Test.java

This class gets the bean from the applicationContext.xml file and calls the displayInfo() method.

```
1.  import org.springframework.beans.factory.BeanFactory;
2.  import org.springframework.beans.factory.xml.XmlBeanFactory;
3.  import org.springframework.core.io.*;
4.  public class Test {
5.  public static void main(String[] args) {
6.     Resource r=new ClassPathResource("applicationContext.xml");
7.     BeanFactory factory=new XmlBeanFactory(r);
8.     Question q=(Question)factory.getBean("q");
9.     q.displayInfo();
10. } }
```

Setter Injection with Map Example (Non-String)

- In this example, we are using **map** as the answer that have Answer and User. Here, we are using key and value pair both as an object. Answer has its own information such as answerId, answer and postedDate, User has its own information such as userId, username, emailId.
- Like previous examples, it is the example of forum where **one question can have multiple answers**.

Question.java

This class contains three properties, getters & setters and displayInfo() method to display the information.

```
1. import java.util.*;
2. public class Question {
3. private int id;
4. private String name;
5. private Map<Answer,User> answers;
6. //getters and setters
7. public void displayInfo(){
8.     System.out.println("question id:"+id);
9.     System.out.println("question name:"+name);
10.    System.out.println("Answers....");
11.    Set<Entry<Answer, User>> set=answers.entrySet();
12.    Iterator<Entry<Answer, User>> itr=set.iterator();
13.    while(itr.hasNext()){
14.        Entry<Answer, User> entry=itr.next();
15.        Answer ans=entry.getKey();
16.        User user=entry.getValue();
17.        System.out.println("Answer Information:");
18.        System.out.println(ans);
19.        System.out.println("Posted By:");
20.        System.out.println(user);
21.    } } }
```

Answer.java

```
1. import java.util.Date;
2. public class Answer {
3. private int id;
4. private String answer;
5. private Date postedDate;
6. public Answer() {}
7. public Answer(int id, String answer, Date postedDate) {
8.     super();
9.     this.id = id;
10.    this.answer = answer;
11.    this.postedDate = postedDate;  }
12. public String toString(){
13.    return "Id:"+id+" Answer:"+answer+" Posted Date:"+postedDate;  }  }
```

User.java

```
1. public class User {
2. private int id;
3. private String name,email;
4. public User() {}
5. public User(int id, String name, String email) {
6.     super();
7.     this.id = id;
8.     this.name = name;
9.     this.email = email;  }
10. public String toString(){
11.    return "Id:"+id+" Name:"+name+" Email Id:"+email;
12. }  }
```

applicationContext.xml

The **key-ref** and **value-ref** attributes of entry **element** is used to define the reference of bean in the map.

```
1. <bean id="answer1" class="com.javatpoint.Answer">
2. <property name="id" value="1"></property>
3. <property name="answer" value="Java is a Programming Language"></property>
4. <property name="postedDate" value="12/12/2001"></property>
5. </bean>
6. <bean id="answer2" class="com.javatpoint.Answer">
7. <property name="id" value="2"></property>
8. <property name="answer" value="Java is a Platform"></property>
9. <property name="postedDate" value="12/12/2003"></property>
10. </bean>
11. <bean id="user1" class="com.javatpoint.User">
12. <property name="id" value="1"></property>
13. <property name="name" value="Arun Kumar"></property>
14. <property name="email" value="arun@gmail.com"></property>
15. </bean>
16. <bean id="user2" class="com.javatpoint.User">
17. <property name="id" value="2"></property>
18. <property name="name" value="Varun Kumar"></property>
19. <property name="email" value="Varun@gmail.com"></property>
20. </bean>
21.  <bean id="q" class="com.javatpoint.Question">
22. <property name="id" value="1"></property>
23. <property name="name" value="What is Java?"></property>
24. <property name="answers">
25. <map>
26. <entry key-ref="answer1" value-ref="user1"></entry>
27. <entry key-ref="answer2" value-ref="user2"></entry>
28. </map>  </property>  </bean>
```

Test.java

This class gets the bean from the applicationContext.xml file and calls the displayInfo() method to display the information.

```
1. import org.springframework.beans.factory.BeanFactory;
2. import org.springframework.beans.factory.xml.XmlBeanFactory;
3. import org.springframework.core.io.*;
4. public class Test {
5. public static void main(String[] args) {
6.     Resource r=new ClassPathResource("applicationContext.xml");
7.     BeanFactory factory=new XmlBeanFactory(r);
8.     Question q=(Question)factory.getBean("q");
9.     q.displayInfo();  }  }
```

Difference between constructor and setter injection

There are many key differences between constructor injection and setter injection.

1. **Partial dependency**: can be injected using setter injection but it is not possible by constructor. Suppose there are 3 properties in a class, having 3 arg constructor and setters methods. In such case, if you want to pass information for only one property, it is possible by setter method only.
2. **Overriding**: Setter injection overrides the constructor injection. If we use both constructor and setter injection, IOC container will use the setter injection.
3. **Changes**: We can easily change the value by setter injection. It doesn't create a new bean instance always like constructor. So setter injection is flexible than constructor injection.

Setter Injection	Constructor Injection
In Setter Injection, partial injection of dependencies can possible, means if we have 3 dependencies like int, string, long, then its not necessary to inject all values if we use setter injection. If you are not inject it will takes default values for those primitives	In constructor injection, partial injection of dependencies cannot possible, because for calling constructor we must pass all the arguments right, if not so we may get error

Setter Injection will overrides the constructor injection value, provided if we write setter and constructor injection for the same property	But, constructor injection cannot overrides the setter injected values
If we have more dependencies for example 15 to 20 are there in our bean class then, in this case setter injection is not recommended as we need to write almost 20 setters right, bean length will increase.	In this case, Constructor injection is highly recommended, as we can inject all the dependencies with in 3 to 4 lines
Setter injection makes bean class object as mutable [We can change]	Constructor injection makes bean class object as immutable [We cannot change]

Autowiring in Spring

- Autowiring feature of spring framework enables you to inject the object dependency implicitly. It internally uses setter or constructor injection.
- Autowiring can't be used to inject primitive and string values. It works with reference only.

Advantage of Autowiring

- It requires the less code because we don't need to write the code to inject the dependency explicitly.

Disadvantage of Autowiring

- No control of programmer.
- It can't be used for primitive and string values.

Autowiring Modes

There are many autowiring modes:

Mode	Description
no	It is the default autowiring mode. It means no autowiring bydefault.
byName	The byName mode injects the object dependency according to name of the bean. In such case, property name and bean name must be same.
byType	The byType mode injects the object dependency according to type. So property name and bean name can be different. It internally calls setter method.
constructor	The constructor mode injects the dependency by calling the constructor of the class. It calls the constructor having large number of parameters.
autodetect	It is deprecated since Spring 3.

Example of Autowiring

- Let's see the simple code to use autowiring in spring. You need to use autowire attribute of bean element to apply the autowire modes.

```
<bean id="a" class=" A" autowire="byName"> </bean>
```

Let's see the full example of autowiring in spring. To create this example,

1. **B.java**
2. **A.java**
3. **applicationContext.xml**
4. **Test.java**

B.java

This class contains a constructor and method only.

```
1. public class B {
2. B( ){System.out.println("b is created");}
3. void print( ){System.out.println("hello b");}  }
```

A.java

This class contains reference of B class and constructor and method.

```
1. public class A {
2. B b;
3. A(){System.out.println("a is created");}
4. public B getB() {
5.     return b; }
6. public void setB(B b) {  this.b = b;  }
7. void print(){System.out.println("hello a");}
8. void display(){
9.     print();
10.    b.print();  }  }
```

applicationContext.xml

```
1. <bean id="b" class="B"></bean>
2. <bean id="a" class="A" autowire="byName"></bean>
3. </beans>
```

Test.java

This class gets the bean from the applicationContext.xml file and calls the display method.

```
1. import org.springframework.context.ApplicationContext;
2. import org.springframework.context.support.ClassPathXmlApplicationContext;
3. public class Test {
4. public static void main(String[] args)
5. {  ApplicationContext context=new ClassPathXmlApplicationContext("applicationContext.xml");
6. A a=context.getBean("a",A.class);
7. a.display();  }  }
```

Output:

```
b is created
a is created
hello a
hello b
```

1. byName autowiring mode

In case of byName autowiring mode, bean id and reference name must be same.

It internally uses setter injection.

```
1. <bean id="b" class="B"></bean>
2. <bean id="a" class="A" autowire="byName"></bean>
```

But, if you change the name of bean, it will not inject the dependency.

Let's see the code where we are changing the name of the bean from b to b1.

```
1. <bean id="b1" class="B"></bean>
2. <bean id="a" class="A" autowire="byName"></bean>
```

2. byType autowiring mode

In case of byType autowiring mode, bean id and reference name may be different. But there must be only one bean of a type. It internally uses setter injection.

```
1. <bean id="b1" class="B"></bean>
2. <bean id="a" class="A" autowire="byType"></bean>
```

In this case, it works fine because you have created an instance of B type.

It doesn't matter that you have different bean name than reference name.

But, if you have multiple bean of one type, it will not work and throw exception.

```
1. <bean id="b1" class="B"></bean>
2. <bean id="b2" class="B"></bean>
3. <bean id="a" class="A" autowire="byName"></bean>
```

3. constructor autowiring mode

In case of constructor autowiring mode, spring container injects the dependency by highest parameterized constructor.

If you have 3 constructors in a class, zero-arg, one-arg and two-arg then injection will be performed by calling the two-arg constructor.

```
1. <bean id="b" class="B"></bean>
2. <bean id="a" class="A" autowire="constructor"></bean>
```

4. no autowiring mode

In case of no autowiring mode, spring container doesn't inject the dependency by autowiring.

```
1. <bean id="b" class="org.sssit.B"></bean>
2. <bean id="a" class="org.sssit.A" autowire="no"></bean>
```

Dependency Injection with Factory Method in Spring

- Spring framework provides facility to inject bean using factory method. To do so, we can use two attributes of bean element.

1. **factory-method:** represents the factory method that will be invoked to inject the bean.
2. **factory-bean:** represents the reference of the bean by which factory method will be invoked. It is used if factory method is non-static.

2 – Vaishalinagar
Nr. Amrapali Railway Crossing
Raiya Road
Rajkot – 360001
Ph.No–(0281)2440478,2472590

3 – Vaishalinagar
Nr. Amrapali Railway Crossing
Raiya Road
Rajkot - 360001
Ph.No–(0281)2471645

A method that returns instance of a class is called **factory method**.

```
1. public class A {
2. public static A getA(){//factory method
3.     return new A();
4. } }
```

Factory Method Types

There can be three types of factory method:

1) A **static factory method** that returns instance of **its own** class. It is used in singleton design pattern.

```
<bean id="a" class="A" factory-method="getA"></bean>
```

2) A **static factory method** that returns instance of **another** class. It is used instance is not known and decided at runtime.

```
<bean id="b" class="A" factory-method="getB"></bean>
```

3) A **non-static factory** method that returns instance of **another** class. It is used instance is not known and decided at runtime.

```
<bean id="a" class="A"></bean>
<bean id="b" class="A" factory-method="getB" factory-bean="a"></bean>
```

Type 1

- Let's see the simple code to inject the dependency by static factory method.

```
<bean id="a" class="A" factory-method="getA"></bean>
```

- Let's see the full example to inject dependency using factory method in spring. To create this example, we have created 3 files.
  1. **A.java**
  2. **applicationContext.xml**
  3. **Test.java**

A.java

This class is a singleton class.

```
1. public class A {
2. private static final A obj=new A();
3. private A(){System.out.println("private constructor");}
4. public static A getA(){
5.     System.out.println("factory method ");
6.     return obj;  }
7. public void msg(){
8.     System.out.println("hello user");  }  }
```

applicationContext.xml

```
1. <bean id="a" class="A" factory-method="getA"></bean>
```

Test.java

This class gets the bean from the applicationContext.xml file and calls the msg method.

```
1. import org.springframework.context.ApplicationContext;
2. import org.springframework.context.support.ClassPathXmlApplicationContext;
3. public class Test {
4. public static void main(String[] args) {
5.     ApplicationContext context=new ClassPathXmlApplicationContext("applicationContext
   .xml");
6.     A a=(A)context.getBean("a");
7.     a.msg();
8. }
9. }
```

Output:

```
private constructor
factory method
hello user
```

Type 2

- Let's see the simple code to inject the dependency by static factory method that returns the instance of another class.
- To create this example, we have created 6 files.
  1. **Printable.java**
  2. **A.java**
  3. **B.java**
  4. **PrintableFactory.java**
  5. **applicationContext.xml**
  6. **Test.java**

Printable.java

```
1. public interface Printable {
2. void print();  }
```

A.java

```
1. public class A implements Printable{
2.     @Override
3.     public void print() {
4.         System.out.println("hello a");
5.     }    }
```

B.java

```
1. public class B implements Printable{
2.     @Override
3.     public void print() {
4.         System.out.println("hello b");
5.     } }
```

PrintableFactory.java

```
1. public class PrintableFactory {
2. public static Printable getPrintable(){
3.     //return new B();
4.           return new A();//return any one instance, either A or B  }  }
```

applicationContext.xml

```
1. <bean id="p" class="PrintableFactory" factory-method="getPrintable"></bean>
```

Test.java

This class gets the bean from the applicationContext.xml file and calls the print() method.

```
1. import org.springframework.context.ApplicationContext;
2. import org.springframework.context.support.ClassPathXmlApplicationContext;
3. public class Test {
4. public static void main(String[] args) {
5.     ApplicationContext context=new ClassPathXmlApplicationContext("applicationContext.xml");
6.     Printable p=(Printable)context.getBean("p");
7.     p.print();
8. }
9. }
```

Output:

```
hello a
```

Type 3

- Let's see the example to inject the dependency by non-static factory method that returns the instance of another class.
- To create this example, we have created 6 files.
- All files are same as previous, you need to change only 2 files: PrintableFactory and applicationContext.xml.

1. **Printable.java**
2. **A.java**
3. **B.java**
4. **PrintableFactory.java**
5. **applicationContext.xml**
6. **Test.java**

PrintableFactory.java

```
1. public class PrintableFactory {
2. //non-static factory method
3. public Printable getPrintable(){
4.     return new A();//return any one instance, either A or B
5. } }
```

applicationContext.xml

```
1. <bean id="pfactory" class=" PrintableFactory"></bean>
2. <bean id="p" class=" PrintableFactory" factory-method="getPrintable"
3. factory-bean="pfactory"></bean>
```

Output:

```
hello a
```

Spring AOP Tutorial

- **Aspect Oriented Programming** (AOP) compliments OOPs in the sense that it also provides modularity. But the key unit of modularity is aspect than class.
- AOP breaks the program logic into distinct parts (called concerns). It is used to increase modularity by **cross-cutting concerns**.
- A **cross-cutting concern** is a concern that can affect the whole application and should be centralized in one location in code as possible, such as transaction management, authentication, logging, security etc.

Why use AOP?

- It provides the pluggable way to dynamically add the additional concern before, after or around the actual logic. Suppose there are 10 methods in a class as given below:

```
1. class A{
2. public void m1(){...}
3. public void m2(){...}
4. public void m3(){...}
5. public void m4(){...}
6. public void m5(){...}
7. public void n1(){...}
```

```
8.  public void n2(){...}
9.  public void p1(){...}
10. public void p2(){...}
11. public void p3(){...}
12. }
```

- There are 5 methods that starts from m, 2 methods that starts from n and 3 methods that starts from p.
- **Understanding Scenario** I have to maintain log and send notification after calling methods that starts from m.
- **Problem without AOP** We can call methods (that maintains log and sends notification) from the methods starting with m. In such scenario, we need to write the code in all the 5 methods.
- But, if client says in future, I don't have to send notification, you need to change all the methods. It leads to the maintenance problem.
- **Solution with AOP** We don't have to call methods from the method. Now we can define the additional concern like maintaining log, sending notification etc. in the method of a class. Its entry is given in the xml file.
- In future, if client says to remove the notifier functionality, we need to change only in the xml file. So, maintenance is easy in AOP.

Where use AOP?

AOP is mostly used in following cases:

- To provide declarative enterprise services such as declarative transaction management.
- It allows users to implement custom aspects.

AOP Concepts and Terminology

AOP concepts and terminologies are as follows:

- Join point
- Advice
- Pointcut
- Introduction
- Target Object
- Aspect
- Interceptor
- AOP Proxy
- Weaving

Join point

- Join point is any point in your program such as method execution, exception handling, field access etc. Spring supports only method execution join point.

Advice

- Advice represents an action taken by an aspect at a particular join point. There are different types of advices:
    - **Before Advice**: it executes before a join point.
    - **After Returning Advice:** it executes after a joint point completes normally.
    - **After Throwing Advice:** it executes if method exits by throwing an exception.
    - **After (finally) Advice:** it executes after a join point regardless of join point exit whether normally or exceptional return.
    - **Around Advice:** It executes before and after a join point.

Pointcut

- It is an expression language of AOP that matches join points.

Introduction

- It means introduction of additional method and fields for a type. It allows you to introduce new interface to any advised object.

Target Object

- It is the object i.e. being advised by one or more aspects. It is also known as proxied object in spring because Spring AOP is implemented using runtime proxies.

Aspect

- It is a class that contains advices, joinpoints etc.

Interceptor

- It is an aspect that contains only one advice.

AOP Proxy

- It is used to implement aspect contracts, created by AOP framework. It will be a JDK dynamic proxy or CGLIB proxy in spring framework.

Weaving

- It is the process of linking aspect with other application types or objects to create an advised object. Weaving can be done at compile time, load time or runtime. Spring AOP performs weaving at runtime.

AOP Implementations

AOP implementations are provided by:

1. AspectJ
2. Spring AOP
3. JBoss AOP

Spring AOP with Example

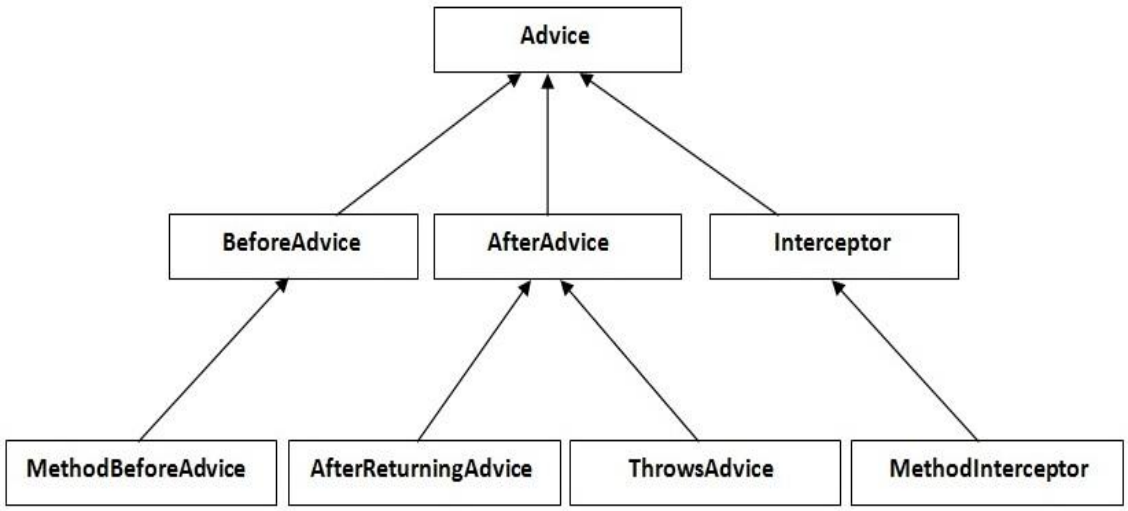
Question:

1) Before Advice Example
2) After Returning Advice Example
3) Around Advice Example
4) After Throwing Advice Example

- Given examples are of **Spring1.2 old style AOP** implementation.
- Though it is supported in spring 3, but it is recommended to use spring aop with aspectJ that we are going to learn in next page.

There are 4 types of advices supported in spring1.2 old style aop implementation.

1. **Before Advice** it is executed before the actual method call.
2. **After Advice** it is executed after the actual method call. If method returns a value, it is executed after returning value.
3. **Around Advice** it is executed before and after the actual method call.
4. **Throws Advice** it is executed if actual method throws exception.



Understanding the hierarchy of advice interfaces

- All are interfaces in AOP.
- Let's understand the advice hierarchy by the diagram given below:

1) MethodBeforeAdvice Example

- Create a class that contains actual business logic.

A.java

```
1. public class A {
2. public void m(){System.out.println("actual business logic");}  }
```

- Now, create the advisor class that **implements MethodBeforeAdvice interface.**

BeforeAdvisor.java

```
1. import java.lang.reflect.Method;
2. import org.springframework.aop.MethodBeforeAdvice;
3. public class BeforeAdvisor implements MethodBeforeAdvice{
4.     @Override
5.     public void before(Method method, Object[] args, Object target)throws Throwable {
   System.out.println("additional concern before actual logic");     } }
```

- In xml file, create 3 beans, one for A class, second for Advisor class and third for **ProxyFactoryBean** class.

applicationContext.xml

```
1. <bean id="obj" class=" A"></bean>
2. <bean id="ba" class=" BeforeAdvisor"></bean>
3.  <bean id="proxy" class="org.springframework.aop.framework.ProxyFactoryBean"> <pr
   operty name="target" ref="obj"></property>
4. <property name="interceptorNames">
5. <list> <value>ba</value>
6. </list> </property> </bean>
```

Understanding ProxyFactoryBean class:

- The ProxyFactoryBean class is provided by Spring Famework.
- It contains 2 properties target and interceptorNames.
- The instance of A class will be considered as target object and the instance of advisor class as interceptor.
- You need to pass the advisor object as the list object as in the xml file given above.
- The ProxyFactoryBean class is written something like this:

```
1. public class ProxyFactoryBean{
2. private Object target;
3. private List interceptorNames;
4. //getters and setters  }
```

- Now, let's call the actual method.

Test.java

```
1. import org.springframework.beans.factory.BeanFactory;
2. import org.springframework.beans.factory.xml.XmlBeanFactory;
3. import org.springframework.core.io.ClassPathResource;
4. import org.springframework.core.io.Resource;
5. public class Test {
6. public static void main(String[] args) {
7.     Resource r=new ClassPathResource("applicationContext.xml");
8.     BeanFactory factory=new XmlBeanFactory(r);
9.    A a=factory.getBean("proxy",A.class);
10.    a.m();
11. } }
```

Output

```
additional concern before actual logic
actual business logic
```

2) AfterReturningAdvice Example

- Create a class that contains actual business logic.

A.java

- **Same as in the previous example.**
- Now, create the advisor class that implements AfterReturningAdvice interface.

AfterAdvisor.java

```
1. import java.lang.reflect.Method;
2. import org.springframework.aop.AfterReturningAdvice;
3. public class AfterAdvisor implements AfterReturningAdvice{
4.    @Override
5.     public void afterReturning(Object returnValue, Method method,
6.          Object[] args, Object target) throws Throwable {
7.         System.out.println("additional concern after returning advice");
8.     } }
```

- Create the xml file as in the previous example, you need to change only the advisor class here.

applicationContext.xml

```
1. <bean id="obj" class="A"></bean>
2. <bean id="ba" class="AfterAdvisor"></bean>
3.    <bean id="proxy" class="org.springframework.aop.framework.ProxyFactoryBean">
4. <property name="target" ref="obj"></property>
5. <property name="interceptorNames">
6. <list>
7. <value>ba</value>
8. </list>
9. </property>
10. </bean>
```

Test.java

Same as in the previous example.

Output

```
actual business logic
additional concern after returning advice
```

3) MethodInterceptor (AroundAdvice) Example

- Create a class that contains actual business logic.

A.java

- **Same as in the previous example.**
- Now, create the advisor class that implements MethodInterceptor interface.

AroundAdvisor.java

```
1. import org.aopalliance.intercept.MethodInterceptor;
2. import org.aopalliance.intercept.MethodInvocation;
3. public class AroundAdvisor implements MethodInterceptor{
4.  @Override
5.  public Object invoke(MethodInvocation mi) throws Throwable {
6.  Object obj;
7.  System.out.println("additional concern before actual logic");
8.        obj=mi.proceed();
9.        System.out.println("additional concern after actual logic");
10.       return obj;
11.   }  }
```

- Create the xml file as in the previous example, you need to change only the advisor class here.

applicationContext.xml

```
1. <bean id="obj" class="A"></bean>
2. <bean id="ba" class="AroundAdvisor"></bean>
3.   <bean id="proxy" class="org.springframework.aop.framework.ProxyFactoryBean">
4. <property name="target" ref="obj"></property>
5. <property name="interceptorNames">
6. <list>
7. <value>ba</value>
8. </list>
9. </property>
10. </bean>
```

Test.java

- Same as in the previous example.

Output

```
additional concern before actual logic
actual business logic
additional concern after actual logic
```

4) ThrowsAdvice Example

- Create a class that contains actual business logic.

Validator.java

```
1. public class Validator {
2.     public void validate(int age)throws Exception{
3.         if(age<18){
4.             throw new ArithmeticException("Not Valid Age");   }
5.         else{
6.             System.out.println("vote confirmed");   }
7.     }
8. }
```

- Now, create the advisor class that implements ThrowsAdvice interface.

ThrowsAdvisor.java

```
1. import org.springframework.aop.ThrowsAdvice;
2. public class ThrowsAdvisor implements ThrowsAdvice{
3.     public void afterThrowing(Exception ex){
4.         System.out.println("additional concern if exception occurs");
5.     } }
```

- Create the xml file as in the previous example, you need to change only the Validator class and advisor class.

applicationContext.xml

```
1. <bean id="obj" class="Validator"></bean>
2. <bean id="ba" class="ThrowsAdvisor"></bean>
3. <bean id="proxy" class="org.springframework.aop.framework.ProxyFactoryBean">
4. <property name="target" ref="obj"></property>
5. <property name="interceptorNames">
6. <list>
```

```
7. <value>ba</value>
8. </list>
9. </property>
10. </bean>
```

Test.java

```
1. import org.springframework.beans.factory.BeanFactory;
2. import org.springframework.beans.factory.xml.XmlBeanFactory;
3. import org.springframework.core.io.ClassPathResource;
4. import org.springframework.core.io.Resource;
5. public class Test {
6. public static void main(String[] args) {
7.     Resource r=new ClassPathResource("applicationContext.xml");
8.     BeanFactory factory=new XmlBeanFactory(r);
9.     Validator v=factory.getBean("proxy",Validator.class);
10.    try{
11.    v.validate(12);
12.    }catch(Exception e){e.printStackTrace();}
13. } }
```

Spring AOP AspectJ Annotation Example

1. @Before Example
2. @After Example
3. @AfterReturning Example
4. @Around Example
5. @AfterThrowing Example

- The Spring Framework recommends you to use Spring AspectJ AOP implementation over the Spring 1.2 old style dtd based AOP implementation because it provides you more control and it is easy to use.
- There are two ways to use Spring AOP AspectJ implementation:
  1. By annotation
  2. By xml configuration (schema based)
- Spring AspectJ AOP implementation provides many annotations:
  1. **@Aspect** declares the class as aspect.

2. **@Pointcut** declares the pointcut expression.

- The annotations used to create advices are given below:
  1. **@Before** declares the before advice. It is applied before calling the actual method.
  2. **@After** declares the after advice. It is applied after calling the actual method and before returning result.
  3. **@AfterReturning** declares the after returning advice. It is applied after calling the actual method and before returning result. But you can get the result value in the advice.
  4. **@Around** declares the around advice. It is applied before and after calling the actual method.
  5. **@AfterThrowing** declares the throws advice. It is applied if actual method throws exception.

Understanding Pointcut

- Pointcut is an expression language of Spring AOP.
- The **@Pointcut** annotation is used to define the pointcut. We can refer the pointcut expression by name also. Let's see the simple example of pointcut expression.

```
1. @Pointcut("execution(* Operation.*(..))")
2. private void doSomething() {}
```

- The name of the pointcut expression is doSomething(). It will be applied on all the methods of Operation class regardless of return type.

Understanding Pointcut Expressions

- Let's try the understand the pointcut expressions by the examples given below:

```
1. @Pointcut("execution(public * *(..))")
```

- It will be applied on all the public methods.

```
2. @Pointcut("execution(public Operation.*(..))")
```

- It will be applied on all the public methods of Operation class.

```
3. @Pointcut("execution(* Operation.*(..))")
```

- It will be applied on all the methods of Operation class.

```
4. @Pointcut("execution(public Employee.set*(..))")
```

- It will be applied on all the public setter methods of Employee class.

```
5. @Pointcut("execution(int Operation.*(..))")
```

- It will be applied on all the methods of Operation class that returns int value.

1) @Before Example

The AspectJ Before Advice is applied before the actual business logic method.

You can perform any operation here such as conversion, authentication etc.

Create a class that contains actual business logic.

Operation.java

```
1. public  class Operation{
2.     public void msg(){System.out.println("msg method invoked");}
3.     public int m(){System.out.println("m method invoked");return 2;}
4.     public int k(){System.out.println("k method invoked");return 3;}
5. }
```

- Now, create the aspect class that contains before advice.

TrackOperation.java

```
1. import org.aspectj.lang.JoinPoint;
2. import org.aspectj.lang.annotation.Aspect;
3. import org.aspectj.lang.annotation.Before;
4. import org.aspectj.lang.annotation.Pointcut;
5. @Aspect
6. public class TrackOperation{
7.     @Pointcut("execution(* Operation.*(..))")
8.     public void k(){}//pointcut name
9.         @Before("k()")//applying pointcut on before advice
10.    public void myadvice(JoinPoint jp)//it is advice (before advice)
11.    {    System.out.println("additional concern");
12.       //System.out.println("Method Signature: "  + jp.getSignature());
13.    } }
```

- Now create the applicationContext.xml file that defines beans.

applicationContext.xml

```
1.     <bean id="opBean" class="com.javatpoint.Operation">   </bean>
2.     <bean id="trackMyBean" class="com.javatpoint.TrackOperation"></bean>
3.       <bean class="org.springframework.aop.aspectj.annotation.AnnotationAwareAspect
   JAutoProxyCreator"></bean>
```

2 – Vaishalinagar
Nr. Amrapali Railway Crossing
Raiya Road
Rajkot – 360001
Ph.No–(0281)2440478,2472590

3 – Vaishalinagar
Nr. Amrapali Railway Crossing
Raiya Road
Rajkot - 360001
Ph.No–(0281)2471645

Test.java

```
1. import org.springframework.context.ApplicationContext;
2. import org.springframework.context.support.ClassPathXmlApplicationContext;
3. public class Test{
4. public static void main(String[] args){
5. ApplicationContext context = new ClassPathXmlApplicationContext("applicationCont
   ext.xml");
6. Operation e = (Operation) context.getBean("opBean");
7.  System.out.println("calling msg...");
8.  e.msg();
9.  System.out.println("calling m...");
10.  e.m();
11. System.out.println("calling k...");
12.  e.k();
13.    }
14. }
```

Output

```
calling msg...
additional concern
msg() method invoked
calling m...
additional concern
m() method invoked
calling k...
additional concern
k() method invoked
```

- As you can see, additional concern is printed before msg(), m() and k() method is invoked.
- Now if you change the pointcut expression as given below:

```
1. @Pointcut("execution(* Operation.m*(..))")
```

- Now additional concern will be applied for the methods starting with m in Operation class.

- Output will be as this:

```
calling msg...
additional concern
msg() method invoked
calling m...
additional concern
m() method invoked
calling k...
k() method invoked
```

2) @After Example

- The AspectJ after advice is applied after calling the actual business logic methods. It can be used to maintain log, security, notification etc.
- Here, We are assuming that **Operation.java**, **applicationContext.xml** and **Test.java** files are same as given in @Before example.
- Create the aspect class that contains after advice.

TrackOperation.java

```
1. import org.aspectj.lang.JoinPoint;
2. import org.aspectj.lang.annotation.Aspect;
3. import org.aspectj.lang.annotation.After;
4. import org.aspectj.lang.annotation.Pointcut;
5. @Aspect
6. public class TrackOperation{
7.     @Pointcut("execution(* Operation.*(..))")
8.     public void k(){}//pointcut name
9.     @After("k()")//applying pointcut on after advice
10.    public void myadvice(JoinPoint jp)//it is advice (after advice)
11.    {
12.        System.out.println("additional concern");
13.        //System.out.println("Method Signature: "  + jp.getSignature());
14.    } }
```

Output

```
calling msg...
msg() method invoked
additional concern
calling m...
m() method invoked
additional concern
calling k...
k() method invoked
additional concern
```

3) @AfterReturning Example

- By using after returning advice, we can get the result in the advice.
- Create the class that contains business logic.

Operation.java

```
1. public  class Operation{
2.     public int m(){System.out.println("m() method invoked");return 2;}
3.     public int k(){System.out.println("k() method invoked");return 3;}
4. }
```

Create the aspect class that contains after returning advice.

TrackOperation.java

```
1. import org.aspectj.lang.JoinPoint;
2. import org.aspectj.lang.annotation.*;
3. @Aspect
4. public class TrackOperation{
5.     @AfterReturning(
6.             pointcut = "execution(* Operation.*(..))",
7.             returning= "result")
8.     public void myadvice(JoinPoint jp,Object result)
9.     {   System.out.println("additional concern");
10.       System.out.println("Method Signature: "  + jp.getSignature());
11.       System.out.println("Result in advice: "+result);
12.       System.out.println("end of after returning advice...");
13.    } }
```

applicationContext.xml

- It is same as given in @Before advice example

Test.java

- Now create the Test class that calls the actual methods.

```
1. import org.springframework.context.ApplicationContext;
2. import org.springframework.context.support.*;
3. public class Test{
4.     public static void main(String[] args){
5. ApplicationContext context = new ClassPathXmlApplicationContext("applicationCont
   ext.xml");
6.         Operation e = (Operation) context.getBean("opBean");
7.         System.out.println("calling m...");
8.         System.out.println(e.m());
9.         System.out.println("calling k...");
10.        System.out.println(e.k());      }}
```

Output

```
calling m...
m() method invoked
additional concern
Method Signature: int com.javatpoint.Operation.m()
Result in advice: 2
end of after returning advice...
2
calling k...
k() method invoked
additional concern
Method Signature: int com.javatpoint.Operation.k()
Result in advice: 3
end of after returning advice...
3
```

4) @Around Example

- The AspectJ around advice is applied before and after calling the actual business logic methods.
- Here, we are assuming that **applicationContext.xml** file is same as given in @Before example.
- Create a class that contains actual business logic.

Operation.java

```
1. public  class Operation{
2.     public void msg(){System.out.println("msg() is invoked");}
3.     public void display(){System.out.println("display() is invoked");}
4. }
```

- Create the aspect class that contains around advice.
- You need to pass the **PreceedingJoinPoint** reference in the advice method, so that we can proceed the request by calling the proceed() method.

TrackOperation.java

```
1. import org.aspectj.lang.ProceedingJoinPoint;
2. import org.aspectj.lang.annotation.*;
3.   @Aspect
4. public class TrackOperation
5. {
6.     @Pointcut("execution(* Operation.*(..))")
7.     public void abcPointcut(){}
8.
9.     @Around("abcPointcut()")
10.    public Object myadvice(ProceedingJoinPoint pjp) throws Throwable
11.    {
12.        System.out.println("Additional Concern Before calling actual method");
13.        Object obj=pjp.proceed();
14.        System.out.println("Additional Concern After calling actual method");
15.        return obj;
16.    } }
```

Test.java

- Now create the Test class that calls the actual methods.

```
1. import org.springframework.context.ApplicationContext;
2. import org.springframework.context.support.ClassPathXmlApplicationContext;
3. public class Test{
4.     public static void main(String[] args){
5.         ApplicationContext context = new classPathXmlApplicationContext("application
   Context.xml");
6.         Operation op = (Operation) context.getBean("opBean");
7.         op.msg();
8.         op.display();
9.     } }
```

Output

```
Additional Concern Before calling actual method
msg() is invoked
Additional Concern After calling actual method
Additional Concern Before calling actual method
display() is invoked
Additional Concern After calling actual method
```

5) @AfterThrowing Example

- By using after throwing advice, we can print the exception in the TrackOperation class. Let's see the example of AspectJ AfterThrowing advice.
- Create the class that contains business logic.

Operation.java

```
1. public  class Operation{
2.     public void validate(int age)throws Exception{
3.     if(age<18){
4.         throw new ArithmeticException("Not valid age");   }
5.     else{
6.         System.out.println("Thanks for vote");      }
7.     } }
```

- Create the aspect class that contains after throwing advice.
- Here, we need to pass the Throwable reference also, so that we can intercept the exception here.

TrackOperation.java

```
1. import org.aspectj.lang.JoinPoint;
2. import org.aspectj.lang.annotation.*;
3. @Aspect
4. public class TrackOperation{
5.    @AfterThrowing(
6.            pointcut = "execution(* Operation.*(..))",
7.            throwing= "error")
8.    public void myadvice(JoinPoint jp,Throwable error)//it is advice
9.    {  System.out.println("additional concern");
10.      System.out.println("Method Signature: "  + jp.getSignature());
11.      System.out.println("Exception is: "+error);
12.      System.out.println("end of after throwing advice...");
13.   } }
```

applicationContext.xml

- **It is same as given in @Before advice example**

Test.java

- Now create the Test class that calls the actual methods.

```
1. import org.springframework.context.ApplicationContext;
2. import org.springframework.context.support.ClassPathXmlApplicationContext;
3. public class Test{
4.    public static void main(String[] args){
5. ApplicationContext context = new ClassPathXmlApplicationContext("applicationCont
   ext.xml");
6.       Operation op = (Operation) context.getBean("opBean");
7.       System.out.println("calling validate...");
8.       try{
9.      op.validate(19);
10.      }catch(Exception e){System.out.println(e);}
11.      System.out.println("calling validate again...");
12.      try{
13.      op.validate(11);
14.      }catch(Exception e){System.out.println(e);}
15.   } }
```

Output

```
calling validate...
Thanks for vote
calling validate again...
additional concern
Method Signature: void com.javatpoint.Operation.validate(int)
Exception is: java.lang.ArithmeticException: Not valid age
end of after throwing advice...
java.lang.ArithmeticException: Not valid age
```

Spring AOP AspectJ Xml Configuration Example

1. aop:before example
2. aop:after example
3. aop:after-returning example
4. aop:around example
5. aop:after-throwing example

- Spring enables you to define the aspects, advices and pointcuts in xml file.
- Let's see the xml elements that are used to define advice.

1. **aop:before** It is applied before calling the actual business logic method.
2. **aop:after** It is applied after calling the actual business logic method.
3. **aop:after-returning** it is applied after calling the actual business logic method. It can be used to intercept the return value in advice.
4. **aop:around** It is applied before and after calling the actual business logic method.
5. **aop:after-throwing** It is applied if actual business logic method throws exception.

1) aop:before Example

- The AspectJ Before Advice is applied before the actual business logic method. You can perform any operation here such as conversion, authentication etc.
- Create a class that contains actual business logic.

Operation.java

```
1. public  class Operation{
2.     public void msg(){System.out.println("msg method invoked");}
3.     public int m(){System.out.println("m method invoked");return 2;}
4.     public int k(){System.out.println("k method invoked");return 3;}}
```

- Now, create the aspect class that contains before advice.

TrackOperation.java

```
1. import org.aspectj.lang.JoinPoint;
2. public class TrackOperation{
3.     public void myadvice(JoinPoint jp)//it is advice
4.    {   System.out.println("additional concern");  }  }
```

- Now create the applicationContext.xml file that defines beans.

applicationContext.xml

```
1. <aop:aspectj-autoproxy />
2.   <bean id="opBean" class=" Operation">   </bean>
3. <bean id="trackAspect" class="TrackOperation"></bean>
4.  <aop:config>
5.   <aop:aspect id="myaspect" ref="trackAspect" >
6.      <!-- @Before -->
7. <aop:pointcut id="pointCutBefore"   expression="execution(* Operation.*(..))" />
8.      <aop:before method="myadvice" pointcut-ref="pointCutBefore" />
9.   </aop:aspect>
10. </aop:config>
```

- Now, let's call the actual method.

Test.java

```
1. import org.springframework.context.ApplicationContext;
2. import org.springframework.context.support.ClassPathXmlApplicationContext;
3. public class Test{
4.     public static void main(String[] args){
5.    ApplicationContext context = new ClassPathXmlApplicationContext("applicationCo
   ntext.xml");
6.         Operation e = (Operation) context.getBean("opBean");
7.         System.out.println("calling msg...");
8.         e.msg();
9.         System.out.println("calling m...");
10.        e.m();
11.        System.out.println("calling k...");
12.        e.k();     }  }
```

Output

```
calling msg...
additional concern
msg() method invoked
calling m...
additional concern
m() method invoked
calling k...
additional concern
k() method invoked
```

2) aop:after example

- The AspectJ after advice is applied after calling the actual business logic methods. It can be used to maintain log, security, notification etc.
- Here, We are assuming that **Operation.java**, **TrackOperation.java** and **Test.java** files are same as given in aop:before example.
- Now create the applicationContext.xml file that defines beans.

applicationContext.xml

```
1.  <aop:aspectj-autoproxy />
2.    <bean id="opBean" class=" Operation">   </bean>
3.  <bean id="trackAspect" class=" TrackOperation"></bean>
4.  <aop:config>
5.    <aop:aspect id="myaspect" ref="trackAspect" >
6.       <!-- @After -->
7.   <aop:pointcut id="pointCutAfter"    expression="execution(* Operation.*(..))" />
8.       <aop:after method="myadvice" pointcut-ref="pointCutAfter" />
9.    </aop:aspect>
10. </aop:config>
```

Output

```
calling msg...
msg() method invoked
additional concern
calling m...
m() method invoked
```

```
additional concern
calling k...
k() method invoked
additional concern
```

3) aop:after-returning example

- By using after returning advice, we can get the result in the advice.
- Create the class that contains business logic.

Operation.java

```
1. public  class Operation{
2.     public int m(){System.out.println("m() method invoked");return 2;}
3.     public int k(){System.out.println("k() method invoked");return 3;}
4. }
```

- Create the aspect class that contains after returning advice.

TrackOperation.java

```
1. import org.aspectj.lang.JoinPoint;
2.   public class TrackOperation{
3.     public void myadvice(JoinPoint jp,Object result)
4.     {          System.out.println("additional concern");
5.        System.out.println("Method Signature: "  + jp.getSignature());
6.        System.out.println("Result in advice: "+result);
7.        System.out.println("end of after returning advice...");
8.     }  }
```

applicationContext.xml

```
1. <aop:aspectj-autoproxy />
2. <bean id="opBean" class="com.javatpoint.Operation">   </bean>
3.       <bean id="trackAspect" class="com.javatpoint.TrackOperation"></bean>
4.          <aop:config>
5.   <aop:aspect id="myaspect" ref="trackAspect" >
6.      <!-- @AfterReturning -->
7. <aop:pointcut id="CutAfterReturning" expression="execution(* Operation.*(..))" />
8. <aop:after-returning method="myadvice" returning="result" pointcut-
   ref="CutAfterReturning" />
9.   </aop:aspect>
10. </aop:config>
```

Test.java

- Now create the Test class that calls the actual methods.

```
1. import org.springframework.context.ApplicationContext;
2. import org.springframework.context.support.ClassPathXmlApplicationContext;
3. public class Test{
4.     public static void main(String[] args){
5.  ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.
    xml");
6.         Operation e = (Operation) context.getBean("opBean");
7.         System.out.println("calling m...");
8.         System.out.println(e.m());
9.         System.out.println("calling k...");
10.        System.out.println(e.k());
11.    }
12. }
```

Output

```
1. calling m...
2. m() method invoked
3. additional concern
4. Method Signature: int com.javatpoint.Operation.m()
5. Result in advice: 2
6. end of after returning advice...
7. 2
8. calling k...
9. k() method invoked
10. additional concern
11. Method Signature: int com.javatpoint.Operation.k()
12. Result in advice: 3
13. end of after returning advice...
14. 3
```

4) aop:around example

- The AspectJ around advice is applied before and after calling the actual business logic methods.
- Create a class that contains actual business logic.

2 – Vaishalinagar
Nr. Amrapali Railway Crossing
Raiya Road
Rajkot – 360001
Ph.No–(0281)2440478,2472590

3 – Vaishalinagar
Nr. Amrapali Railway Crossing
Raiya Road
Rajkot - 360001
Ph.No–(0281)2471645

Operation.java

```
1. public  class Operation{
2.     public void msg(){System.out.println("msg() is invoked");}
3.     public void display(){System.out.println("display() is invoked");}
4. }
```

- Create the aspect class that contains around advice.
- You need to pass the PreceedingJoinPoint reference in the advice method, so that we can proceed the request by calling the proceed() method.

TrackOperation.java

```
1. import org.aspectj.lang.ProceedingJoinPoint;
2. public class TrackOperation
3. {  public Object myadvice(ProceedingJoinPoint pjp) throws Throwable
4.     {
5.         System.out.println("Additional Concern Before calling actual method");
6.         Object obj=pjp.proceed();
7.         System.out.println("Additional Concern After calling actual method");
8.         return obj;
9.     }
10. }
```

applicationContext.xml

```
1. <aop:aspectj-autoproxy />
2.   <bean id="opBean" class=" Operation">   </bean>
3.   <bean id="trackAspect" class=" TrackOperation"></bean>
4.    <aop:config>
5.   <aop:aspect id="myaspect" ref="trackAspect" >
6.      <!-- @Around -->
7.      <aop:pointcut id="pointCutAround"   expression="execution(* Operation.*(..))" />
    <aop:around method="myadvice" pointcut-ref="pointCutAround" />
8.   </aop:aspect>
9. </aop:config>
```

2 – Vaishalinagar
Nr. Amrapali Railway Crossing
Raiya Road
Rajkot – 360001
Ph.No–(0281)2440478,2472590

3 – Vaishalinagar
Nr. Amrapali Railway Crossing
Raiya Road
Rajkot - 360001
Ph.No–(0281)2471645

Test.java

- Now create the Test class that calls the actual methods.

```
1. import org.springframework.context.ApplicationContext;
2. import org.springframework.context.support.ClassPathXmlApplicationContext;
3. public class Test{
4.     public static void main(String[] args){
5. ApplicationContext context = new classPathXmlApplicationContext("applicationContext.x
   ml");
6.         Operation op = (Operation) context.getBean("opBean");
7.         op.msg();
8.         op.display();
9.     }
10. }
```

Output

```
Additional Concern Before calling actual method
msg() is invoked
Additional Concern After calling actual method
Additional Concern Before calling actual method
display() is invoked
Additional Concern After calling actual method
```

5) aop:after-throwing example

- By using after throwing advice, we can print the exception in the TrackOperation class. Let's see the example of AspectJ AfterThrowing advice.
- Create the class that contains business logic.

Operation.java

```
1. package com.javatpoint;
2. public  class Operation{
3.     public void validate(int age)throws Exception{
4.     if(age<18){
5.         throw new ArithmeticException("Not valid age");     }
6.     else{         System.out.println("Thanks for vote");     }
7.     }  }
```

- Create the aspect class that contains after throwing advice.

- Here, we need to pass the Throwable reference also, so that we can intercept the exception here.

TrackOperation.java

```
1. import org.aspectj.lang.JoinPoint;
2. public class TrackOperation{
3. public void myadvice(JoinPoint jp,Throwable error)//it is advice
4.     {   System.out.println("additional concern");
5.         System.out.println("Method Signature: "  + jp.getSignature());
6.         System.out.println("Exception is: "+error);
7.         System.out.println("end of after throwing advice...");
8.     } }
```

applicationContext.xml

```
1. <aop:aspectj-autoproxy />
2. <bean id="opBean" class=" Operation">   </bean>
3. <bean id="trackAspect" class="TrackOperation"></bean>
4.  <aop:config>
5.   <aop:aspect id="myaspect" ref="trackAspect" >
6.      <!-- @AfterThrowing -->
7.   <aop:pointcut id="AfterThrowing"  expression="execution(* Operation.*(..))" />
8.      <aop:after-throwing method="myadvice" throwing="error" pointcut-
   ref="AfterThrowing" />
9.   </aop:aspect>
10. </aop:config>
```

Test.java

- Now create the Test class that calls the actual methods.

```
1. import org.springframework.context.ApplicationContext;
2. import org.springframework.context.support.ClassPathXmlApplicationContext;
3. public class Test{
4.     public static void main(String[] args){
5.    ApplicationContext context = new ClassPathXmlApplicationContext("applicationCo
   ntext.xml");
6.         Operation op = (Operation) context.getBean("opBean");
7.         System.out.println("calling validate...");
8.         try{       op.validate(19);
9.         }catch(Exception e){System.out.println(e);}
```

2 – Vaishalinagar
Nr. Amrapali Railway Crossing
Raiya Road
Rajkot – 360001
Ph.No–(0281)2440478,2472590

3 – Vaishalinagar
Nr. Amrapali Railway Crossing
Raiya Road
Rajkot - 360001
Ph.No–(0281)2471645

```
10.        System.out.println("calling validate again...");
11.         try{        op.validate(11);
12.        }catch(Exception e){System.out.println(e);}
13.     }  }
```

Output

```
1.  calling validate...
2.  Thanks for vote
3.  calling validate again...
4.  additional concern
5.  Method Signature: void com.javatpoint.Operation.validate(int)
6.  Exception is: java.lang.ArithmeticException: Not valid age
7.  end of after throwing advice...
8.  java.lang.ArithmeticException: Not valid age
```

Unit - 2

Spring JdbcTemplate Tutorial

- Spring **JdbcTemplate** is a powerful mechanism to connect to the database and execute SQL queries. It internally uses JDBC api, but eliminates a lot of problems of JDBC API.

Problems of JDBC API

The problems of JDBC API are as follows:

- We need to write a lot of code before and after executing the query, such as creating connection, statement, closing resultset, connection etc.
- We need to perform exception handling code on the database logic.
- We need to handle transaction.
- Repetition of all these codes from one to another database logic is a time consuming task.

Advantage of Spring JdbcTemplate

Spring JdbcTemplate eliminates all the above mentioned problems of JDBC API. It provides you methods to write the queries directly, so it saves a lot of work and time.

Spring Jdbc Approaches

Spring framework provides following approaches for JDBC database access:

- JdbcTemplate
- NamedParameterJdbcTemplate
- SimpleJdbcTemplate
- SimpleJdbcInsert and SimpleJdbcCall

JdbcTemplate class

- It is the central class in the Spring JDBC support classes. It takes care of creation and release of resources such as creating and closing of connection object etc. So it will not lead to any problem if you forget to close the connection.
- It handles the exception and provides the informative exception messages by the help of excepion classes defined in the **org.springframework.dao** package.
- We can perform all the database operations by the help of JdbcTemplate class such as insertion, updation, deletion and retrieval of the data from the database.
- Let's see the methods of spring JdbcTemplate class.

No.	Method	Description
1)	public int update(String query)	is used to insert, update and delete records.
2)	public int update(String query, Object... args)	is used to insert, update and delete records using PreparedStatement using given arguments.
3)	public void execute(String query)	is used to execute DDL query.

4)	public T execute(String sql, PreparedStatementCallback action)	executes the query by using PreparedStatement callback.
5)	public T query(String sql, ResultSetExtractor rse)	is used to fetch records using ResultSetExtractor.
6)	public List query(String sql, RowMapper rse)	is used to fetch records using RowMapper.

Example of Spring JdbcTemplate

We are assuming that you have created the following **Employee** table with MySQL.

Column	Type	Size	Constraint
id	number	10	Primary/Optional
name	varchar2	100	
salary	number	10	

Employee.java

- This class contains 3 properties with constructors and setter and getters.

```
1. public class Employee {
2. private int id;
3. private String name;
4. private float salary;
5. //no-arg and parameterized constructors
6. //getters and setters
7. }
```

EmployeeDao.java

- It contains one property jdbcTemplate and three methods saveEmployee(), updateEmployee and deleteEmployee().

```
1. import org.springframework.jdbc.core.JdbcTemplate;
2. public class EmployeeDao {
3. private JdbcTemplate jdbcTemplate;
4. public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
5.     this.jdbcTemplate = jdbcTemplate;  }
6.   public int saveEmployee(Employee e){
7.   String query="insert into employee values(  '"+ e.getId() +"','" + e.getName() +"','"
     +e.getSalary()+"')";
8.     return jdbcTemplate.update(query);  }
9. public int updateEmployee(Employee e){
10.   String query="update employee set
```

```
11.    name="'+e.getName()+"',salary="'+e.getSalary()+"' where id="'+e.getId()+"' ";
12.    return jdbcTemplate.update(query);  }
13.  public int deleteEmployee(Employee e){
14.    String query="delete from employee where id="'+e.getId()+"' ";
15.    return jdbcTemplate.update(query);  }
16.  }
```

applicationContext.xml

- The **DriverManagerDataSource** is used to contain the information about the database such as driver class name, connnection URL, username and password.
- There are a property named **datasource** in the JdbcTemplate class of DriverManagerDataSource type.
- So, we need to provide the reference of DriverManagerDataSource object in the JdbcTemplate class for the datasource property.
- Here, we are using the JdbcTemplate object in the EmployeeDao class, so we are passing it by the setter method but you can use constructor also.

```
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-3.0.xsd">
8.  <bean id="ds" class="org.springframework.jdbc.datasource.DriverManagerDataSour
    ce"
9.  <property name="driverClassName" value="com.mysql.jdbc.Driver" />
10. <property name="url" value="jdbc:mysql://localhost:3306/TestDB" />
11. <property name="username" value="root" />
12. <property name="password" value="root" />
13. </bean>
14.   <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
15. <property name="dataSource" ref="ds"></property>
16. </bean>
17.   <bean id="edao" class="EmployeeDao">
18. <property name="jdbcTemplate" ref="jdbcTemplate"></property>
19. </bean>
20.  </beans>
```

Test.java

- This class gets the bean from the applicationContext.xml file and calls the saveEmployee() method.
- You can also call updateEmployee() and deleteEmployee() method by uncommenting the code as well.

```
1. import org.springframework.context.ApplicationContext;
2. import org.springframework.context.support.ClassPathXmlApplicationContext;
3. public class Test {
4. public static void main(String[] args) {
5.     ApplicationContext ctx=new ClassPathXmlApplicationContext("applicationConte
   xt.xml");
6.     EmployeeDao dao=(EmployeeDao)ctx.getBean("edao");
7.     int status=dao.saveEmployee(new Employee(102,"Piryam",35000));
8.     System.out.println(status);
9.     /*int status=dao.updateEmployee(new Employee(102,"Dhruvi",45000));
10.    System.out.println(status); */
11.    /*Employee e=new Employee();
12.    e.setId(102);
13.    int status=dao.deleteEmployee(e);
14.    System.out.println(status);*/
15. } }
```

PreparedStatement Interface in Spring JdbcTemplate with Example

- We can execute parameterized query using Spring JdbcTemplate by the help of **execute()** method of JdbcTemplate class. To use parameterized query, we pass the instance of **PreparedStatementCallback** in the execute method.
- **Syntax of execute method to use parameterized query**
  1. public T execute(String sql,PreparedStatementCallback<T>);

PreparedStatementCallback interface

- It processes the input parameters and output results. In such case, you don't need to care about single and double quotes.

Method of PreparedStatementCallback interface

It has only one method doInPreparedStatement. Syntax of the method is given below:

1. public T doInPreparedStatement(PreparedStatement ps)throws SQLException, DataAcce ssException

Example of using PreparedStatement in Spring

We are assuming that you have created the **Employee** table with MySQL as given above.

Employee.java

- This class contains 3 properties with constructors and setter and getters.

```
1. public class Employee {
2. private int id;
3. private String name;
4. private float salary;
5. //no-arg and parameterized constructors  and getters and setters  }
```

EmployeeDao.java

- It contains one property jdbcTemplate and one method saveEmployeeByPreparedStatement.
- You must understand the concept of annonymous class to understand the code of the method.

```
1. import java.sql.*;
2. import org.springframework.dao.DataAccessException;
3. import org.springframework.jdbc.core.*;
4. public class EmployeeDao {
5. private JdbcTemplate jdbcTemplate;
6.  public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
7.    this.jdbcTemplate = jdbcTemplate;  }
8.  public Boolean saveEmployeeByPreparedStatement(final Employee e){
9.    String query="insert into employee values(?,?,?)";
10.   return jdbcTemplate.execute(query,new PreparedStatementCallback<Boolean>(
    ){
11.   @Override
12.   public Boolean doInPreparedStatement(PreparedStatement ps)
13.         throws SQLException, DataAccessException {
14.     ps.setInt(1,e.getId());
15.     ps.setString(2,e.getName());
16.     ps.setFloat(3,e.getSalary());
17.     return ps.execute();
18.    }  });
19. }  }
```

applicationContext.xml

```
1. <bean id="ds" class="org.springframework.jdbc.datasource.DriverManagerDataSour
   ce">
2. <property name="driverClassName" value="com.mysql.jdbc.Driver" />
3. <property name="url" value="jdbc:mysql://localhost:3306/TestDB" />
4. <property name="username" value="root" />
5. <property name="password" value="root" />
6. </bean>
7.  <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
8. <property name="dataSource" ref="ds"></property>
```

```
9.  </bean>
10. <bean id="edao" class="EmployeeDao">
11. <property name="jdbcTemplate" ref="jdbcTemplate"></property>
12. </bean>
```

Test.java

```
1.  import org.springframework.context.ApplicationContext;
2.  import org.springframework.context.support.ClassPathXmlApplicationContext;
3.  public class Test {
4.  public static void main(String[] args) {
5.      ApplicationContext ctx=new ClassPathXmlApplicationContext("applicationConte
    xt.xml");
6.      EmployeeDao dao=(EmployeeDao)ctx.getBean("edao");
7.      dao.saveEmployeeByPreparedStatement(new Employee(108,"Pooja",35000));
8.  } }
```

ResultSetExtractor Example | Fetching Records by JdbcTemplate

- We can easily fetch the records from the database using **query()** method of **JdbcTemplate** class where we need to pass the instance of ResultSetExtractor.
- **Syntax of query method using ResultSetExtractor**

```
public T query(String sql,ResultSetExtractor<T> rse)
```

ResultSetExtractor Interface

- **ResultSetExtractor** interface can be used to fetch records from the database. It accepts a ResultSet and returns the list.

Method of ResultSetExtractor interface

- It defines only one method extractData that accepts ResultSet instance as a parameter. Syntax of the method is given below:

```
public T extractData(ResultSet rs)throws SQLException,DataAccessException
```

Example of ResultSetExtractor Interface to show all the records of the table

We are assuming that you have created the **Employee** table with MySQL as given above.

Employee.java

This class contains 3 properties with constructors and setter and getters. It defines one extra method toString().

```
1.  public class Employee {
2.  private int id;
3.  private String name;
4.  private float salary;
5.  //no-arg and parameterized constructors
6.  //getters and setters
```

```
7.  public String toString(){
8.      return id+" "+name+" "+salary;
9.  }  }
```

EmployeeDao.java

It contains on property jdbcTemplate and one method getAllEmployees.

```
1.  import java.sql.*;
2.  import java.util.*;
3.  import org.springframework.dao.DataAccessException;
4.  import org.springframework.jdbc.core.JdbcTemplate;
5.  import org.springframework.jdbc.core.ResultSetExtractor;
6.  public class EmployeeDao {
7.  private JdbcTemplate template;
8.  public void setTemplate(JdbcTemplate template) {
9.  this.template = template;  }
10. public List<Employee> getAllEmployees(){
11.  return template.query("select * from employee",new ResultSetExtractor<List<Emplo
     yee>>(){
12.     @Override
13.      public List<Employee> extractData(ResultSet rs) throws SQLException,
14.              DataAccessException {
15.        List<Employee> list=new ArrayList<Employee>();
16.        while(rs.next()){
17.        Employee e=new Employee();
18.        e.setId(rs.getInt(1));
19.        e.setName(rs.getString(2));
20.        e.setSalary(rs.getInt(3));
21.        list.add(e);
22.        }
23.        return list;
24.        }  });
25.  } }
```

applicationContext.xml

```
1.  <bean id="ds" class="org.springframework.jdbc.datasource.DriverManagerDataSour
    ce
2.  <property name="driverClassName" value="com.mysql.jdbc.Driver" />
3.  <property name="url" value="jdbc:mysql://localhost:3306/TestDB" />
4.  <property name="username" value="root" />
5.  <property name="password" value="root" />  </bean>
6.   <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
```

```
7.  <property name="dataSource" ref="ds"></property>  </bean>
8.    <bean id="edao" class="EmployeeDao">
9.  <property name="jdbcTemplate" ref="jdbcTemplate"></property>  </bean>
```

Test.java

This class gets the bean from the applicationContext.xml file and calls the getAllEmployees() method of EmployeeDao class.

```
1.  import java.util.List;
2.  import org.springframework.context.ApplicationContext;
3.  import org.springframework.context.support.ClassPathXmlApplicationContext;
4.  public class Test {
5.  public static void main(String[] args) {
6.  ApplicationContext ctx=new ClassPathXmlApplicationContext("applicationContext.
    xml");
7.      EmployeeDao dao=(EmployeeDao)ctx.getBean("edao");
8.      List<Employee> list=dao.getAllEmployees();
9.      for(Employee e:list)
10.         System.out.println(e);
11.     }    }
```

RowMapper | Fetching records by Spring JdbcTemplate

- Like ResultSetExtractor, we can use RowMapper interface to fetch the records from the database using **query()** method of **JdbcTemplate** class. In the execute of we need to pass the instance of RowMapper now.
- **Syntax of query method using RowMapper**

```
public T query(String sql,RowMapper<T> rm)
```

RowMapper Interface

- **RowMapper** interface allows to map a row of the relations with the instance of user-defined class.
- It iterates the ResultSet internally and adds it into the collection.
- So we don't need to write a lot of code to fetch the records as ResultSetExtractor.

Advantage of RowMapper over ResultSetExtractor

- RowMapper saves a lot of code becuase it internally adds the data of ResultSet into the collection.

Method of RowMapper interface

- It defines only one method mapRow that accepts ResultSet instance and int as the parameter list. Syntax of the method is given below:

```
public T mapRow(ResultSet rs, int rowNumber)throws SQLException
```

Example of RowMapper Interface to show all the records of the table

- We are assuming that you have created the **Employee** table with MySQL as given above.

Employee.java

This class contains 3 properties with constructors and setter and getters and one extra method toString().

```
1. public class Employee {
2. private int id;
3. private String name;
4. private float salary;
5. //no-arg and parameterized constructors
6. //getters and setters
7. public String toString(){
8.     return id+" "+name+" "+salary;  }  }
```

EmployeeDao.java

It contains on property jdbcTemplate and one method getAllEmployeesRowMapper.

```
1. import java.sql.*;
2. import java.util.*;
3. import org.springframework.dao.DataAccessException;
4. import org.springframework.jdbc.core.*;
5. public class EmployeeDao {
6. private JdbcTemplate template;
7.  public void setTemplate(JdbcTemplate template) {
8.     this.template = template;  }
9.  public List<Employee> getAllEmployeesRowMapper(){
10. return template.query("select * from employee",new RowMapper<Employee>(){
11.    @Override
12.    public Employee mapRow(ResultSet rs, int rownumber) throws SQLException {
13.       Employee e=new Employee();
14.       e.setId(rs.getInt(1));
15.       e.setName(rs.getString(2));
16.       e.setSalary(rs.getInt(3));
17.       return e;
18.    }     });
19. }  }
```

applicationContext.xml

```
1. <bean id="ds" class="org.springframework.jdbc.datasource.DriverManagerDataSour
   ce">
2. <property name="driverClassName" value="com.mysql.jdbc.Driver" />
3. <property name="url" value="jdbc:mysql://localhost:3306/TestDB" />
4. <property name="username" value="root" />
5. <property name="password" value="root" /> </bean>
6. <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
7. <property name="dataSource" ref="ds"></property>  </bean>
8.   <bean id="edao" class="EmployeeDao">
9. <property name="jdbcTemplate" ref="jdbcTemplate"></property>  </bean>
```

Test.java

- This class gets the bean from the applicationContext.xml file and calls the getAllEmployeesRowMapper() method of EmployeeDao class.

```
1. import java.util.List;
2. import org.springframework.context.ApplicationContext;
3. import org.springframework.context.support.ClassPathXmlApplicationContext;
4. public class Test {
5. public static void main(String[] args) {
6.     ApplicationContext ctx=new ClassPathXmlApplicationContext("applicationConte
   xt.xml");
7.     EmployeeDao dao=(EmployeeDao)ctx.getBean("edao");
8.     List<Employee> list=dao.getAllEmployeesRowMapper();
9.     for(Employee e:list)
10.        System.out.println(e);
11. }  }
```

Spring NamedParameter JdbcTemplate with Example

- Spring provides another way to insert data by named parameter.
- In such way, we use names instead of ?(question mark).
- So it is better to remember the data for the column.
- **Simple example of named parameter query**

```
insert into employee values (:id,:name,:salary)
```

- **Method of NamedParameterJdbcTemplate class**
- In this example,we are going to call only the execute method of NamedParameterJdbcTemplate class. Syntax of the method is as follows:

```
pubic T execute(String sql,Map map,PreparedStatementCallback psc)
```

Example of NamedParameterJdbcTemplate class

- We are assuming that you have created the **Employee** table with MySQL as given above.

Employee.java

This class contains 3 properties with constructors and setter and getters.

```
1. public class Employee {
2. private int id;
3. private String name;
4. private float salary;
5. //no-arg and parameterized constructors
6. //getters and setters  }
```

EmployeeDao.java

It contains on property jdbcTemplate and one method save.

```
1. import java.sql.*;
2. import org.springframework.dao.DataAccessException;
3. import org.springframework.jdbc.core.PreparedStatementCallback;
4. import org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate;
5. import java.util.*;
6. public class EmpDao {
7. NamedParameterJdbcTemplate template;
8.   public EmpDao(NamedParameterJdbcTemplate template) {
9.       this.template = template;  }
10. public  void save (Emp e){
11. String query="insert into employee values (:id,:name,:salary)";
12. Map<String,Object> map=new HashMap<String,Object>();
13. map.put("id",e.getId());
14. map.put("name",e.getName());
15. map.put("salary",e.getSalary());
16.  template.execute(query,map,new PreparedStatementCallback() {
17.    @Override
18.    public Object doInPreparedStatement(PreparedStatement ps)
19.          throws SQLException, DataAccessException {
20.       return ps.executeUpdate();
21.    }  });
22. }  }
```

applicationContext.xml

```
1. <bean id="ds" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
2. <property name="driverClassName" value="com.mysql.jdbc.Driver" />
3. <property name="url" value="jdbc:mysql://localhost:3306/TestDB" />
4. <property name="username" value="root" />
```

```
<property name="password" value="root" /> </bean>
<bean id="jtemplate"
 class="org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate">
<constructor-arg ref="ds"></constructor-arg>  </bean>
<bean id="edao" class=" EmpDao">
<constructor-arg>
<ref bean="jtemplate"/>
</constructor-arg>  </bean>
```

SimpleTest.java

This class gets the bean from the applicationContext.xml file and calls the save method.

```
import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.*;
public class SimpleTest {
  public static void main(String[] args) {
    Resource r=new ClassPathResource("applicationContext.xml");
    BeanFactory factory=new XmlBeanFactory(r);
    EmpDao dao=(EmpDao)factory.getBean("edao");
    dao.save(new Emp(23,"Vaishali",50000));
}}
```

Spring SimpleJdbcTemplate Example

- Spring 3 JDBC supports the java 5 feature var-args (variable argument) and autoboxing by the help of SimpleJdbcTemplate class.
- SimpleJdbcTemplate class wraps the JdbcTemplate class and provides the update method where we can pass arbitrary number of arguments.
- **Syntax of update method of SimpleJdbcTemplate class**

```
int update(String sql,Object... parameters)
```

Example of SimpleJdbcTemplate class

- We are assuming that you have created the **Employee** table with MySQL as given above.

Employee.java

Create the class Employee which contains 3 properties with constructors and setter and getters.

EmployeeDao.java

```
import org.springframework.jdbc.core.simple.SimpleJdbcTemplate;
public class EmpDao {
SimpleJdbcTemplate template;
  public EmpDao(SimpleJdbcTemplate template) {  this.template = template;  }
public int update (Emp e){
```

```
6.  String query="update employee set name=? where id=?";
7.  return template.update(query,e.getName(),e.getId());
8.    //String query="update employee set name=?,salary=? where id=?";
9.  //return template.update(query,e.getName(),e.getSalary(),e.getId());
10. }  }
```

applicationContext.xml

```
1.  <bean id="ds" class="org.springframework.jdbc.datasource.DriverManagerDataSource"
2.  <property name="driverClassName" value="com.mysql.jdbc.Driver" />
3.  <property name="url" value="jdbc:mysql://localhost:3306/TestDB" />
4.  <property name="username" value="root" />
5.  <property name="password" value="root" /> </bean>
6.  <bean id="jtemplate"
      class="org.springframework.jdbc.core.simple.SimpleJdbcTemplate">
7.  <constructor-arg ref="ds"></constructor-arg>   </bean>
8.  <bean id="edao" class="EmpDao">
9.  <constructor-arg>
10. <ref bean="jtemplate"/>
11. </constructor-arg>  </bean>
```

SimpleTest.java

This class gets the bean from the applicationContext.xml file and calls the update method of EmpDao class.

```
1.  import org.springframework.beans.factory.BeanFactory;
2.  import org.springframework.beans.factory.xml.XmlBeanFactory;
3.  import org.springframework.core.io.*;
4.  public class SimpleTest {
5.  public static void main(String[] args) {
6.      Resource r=new ClassPathResource("applicationContext.xml");
7.      BeanFactory factory=new XmlBeanFactory(r);
8.      EmpDao dao=(EmpDao)factory.getBean("edao");
9.      int status=dao.update(new Emp(23,"Tarun",35000));
10.     System.out.println(status);
11. }  }
```

Spring with ORM Frameworks

- Spring provides API to easily integrate Spring with ORM frameworks such as Hibernate, JPA(Java Persistence API), JDO(Java Data Objects), Oracle Toplink and iBATIS.

Advantage of ORM Frameworks with Spring

There are a lot of advantage of Spring framework in respect to ORM frameworks. There are as follows:

- **Less coding is required:** By the help of Spring framework, you don't need to write extra codes before and after the actual database logic such as getting the connection, starting transaction, commiting transaction, closing connection etc.
- **Easy to test:** Spring's IoC approach makes it easy to test the application.
- **Better exception handling:** Spring framework provides its own API for exception handling with ORM framework.
- **Integrated transaction management:** By the help of Spring framework, we can wrap our mapping code with an explicit template wrapper class or AOP style method interceptor.

Hibernate and Spring Integration

- We can simply integrate **hibernate application with spring application**.
- In hibernate framework, we provide all the database information hibernate.cfg.xml file.
- But if we are going to integrate the hibernate application with spring, we don't need to create the hibernate.cfg.xml file. We can provide all the information in the applicationContext.xml file.

Advantage of Spring framework with hibernate

- The Spring framework provides **HibernateTemplate** class, so you don't need to follow so many steps like create Configuration, BuildSessionFactory, Session, beginning and committing transaction etc.

Understanding problem without using spring:

Let's understand it by the code of hibernate given below:

```
1. Configuration cfg=new Configuration();
2. cfg.configure("hibernate.cfg.xml");
3. //creating seession factory object
4. SessionFactory factory=cfg.buildSessionFactory();
5. //creating session object
6. Session session=factory.openSession();
7. //creating transaction object
8. Transaction t=session.beginTransaction();
9.  Employee e1=new Employee(111,"arun",40000);
10. session.persist(e1); //persisting the object
11. t.commit(); //transaction is commited
12. session.close();
```

Solution by using HibernateTemplate class of Spring Framework:

Now, you don't need to follow so many steps. You can simply write this:

```
1. Employee e1=new Employee(111,"arun",40000);
2. hibernateTemplate.save(e1);
```

Methods of HibernateTemplate class

Let's see a list of commonly used methods of HibernateTemplate class.

No.	Method	Description
1)	void persist(Object entity)	persists the given object.
2)	Serializable save(Object entity)	persists the given object and returns id.
3)	void saveOrUpdate(Object entity)	persists or updates the given object. If id is found, it updates the record otherwise saves the record.
4)	void update(Object entity)	updates the given object.
5)	void delete(Object entity)	deletes the given object on the basis of id.
6)	Object get(Class entityClass, Serializable id)	returns the persistent object on the basis of given id.
7)	Object load(Class entityClass, Serializable id)	returns the persistent object on the basis of given id.
8)	List loadAll(Class entityClass)	returns the all the persistent objects.

Steps to perform hibernate spring application:

Let's see what are the simple steps for hibernate and spring integration:

1. **create table in the database** It is optional.
2. **create applicationContext.xml file** It contains information of DataSource, SessionFactory etc.
3. **create Employee.java file** It is the persistent class
4. **create employee.hbm.xml file** It is the mapping file.
5. **create EmployeeDao.java file** It is the dao class that uses HibernateTemplate.
6. **create InsertTest.java file** It calls methods of EmployeeDao class.

Example of Hibernate and spring integration

In this example, we are going to integrate the hibernate application with spring. Let's see the **directory structure** of spring and hibernate example.

1. **Create the table in the database**

   Create following table **Employee** in to **MySQL** with database **TestDB**

Column	Type	Size	Constraint
id	number	10,0	Primary key
name	varchar2	255	
salary	Float	126	

2. **Employee.java**

It is a simple POJO class. Here it works as the persistent class for hibernate.

```
1. public class Employee {
2. private int id;
3. private String name;
4. private float salary;
5. //getters and setters
6. }
```

3. **employee.hbm.xml**

This mapping file contains all the information of the persistent class.

```
1. <?xml version='1.0' encoding='UTF-8'?>
2. <!DOCTYPE hibernate-mapping PUBLIC
3. "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
4. "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
5.   <hibernate-mapping>
6. <class name="com.javatpoint.Employee" table="emp558">
7.         <id name="id">
8.         <generator class="assigned"></generator>
9.         </id>
10.        <property name="name"></property>
11.        <property name="salary"></property>
12. </class>
13.  </hibernate-mapping>
```

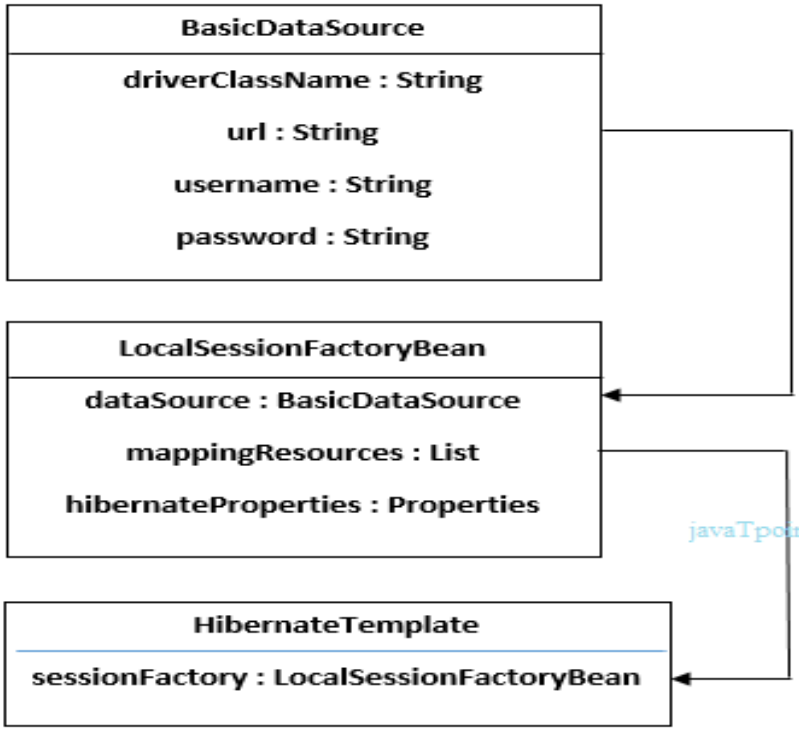
4. **EmployeeDao.java**

It is a java class that uses the **HibernateTemplate** class method to persist the object of Employee class.

```
1. import org.springframework.orm.hibernate3.HibernateTemplate;
2. import java.util.*;
3. public class EmployeeDao {
4. HibernateTemplate template;
5. public void setTemplate(HibernateTemplate template) {
6.     this.template = template;
7. }
8. //method to save employee
9. public void saveEmployee(Employee e){
10.    template.save(e);  }
11. //method to update employee
12. public void updateEmployee(Employee e){
13.    template.update(e);  }
```

```
14. //method to delete employee
15. public void deleteEmployee(Employee e){
16.     template.delete(e);  }
17. //method to return one employee of given id
18. public Employee getById(int id){
19.     Employee e=(Employee)template.get(Employee.class,id);
20.     return e; }
21. //method to return all employees
22. public List<Employee> getEmployees(){
23.     List<Employee> list=new ArrayList<Employee>();
24.     list=template.loadAll(Employee.class);
25.     return list;  }
26. }
```

5. **applicationContext.xml**
   - In this file, we are providing all the informations of the database in the BasicDataSource object.
   - This object is used in the **LocalSessionFactoryBean** class object, containing some other informations such as mappingResources and hibernateProperties.
   - The object of LocalSessionFactoryBean class is used in the HibernateTemplate class.
   - Let's see the code of applicationContext.xml file.



```
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-3.0.xsd">
8.       <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">
9.  <property name="driverClassName"  value=" com.mysql.jdbc.Driver ">
10. </property>
11.  <property name="url" value="jdbc:mysql://localhost:3306/TestDB"></property>
12.       <property name="username" value="root"></property>
13.       <property name="password" value="root"></property>
14.      </bean>
15.      <bean id="mysessionFactory"  class="org.springframework.orm.hibernate3.Local
     SessionFactoryBean">
16.       <property name="dataSource" ref="dataSource"></property>
17.       <property name="mappingResources">
18.       <list>
19.       <value>employee.hbm.xml</value>
20.       </list>
21.       </property>
22.      <property name="hibernateProperties">
23.          <props>
24.              <prop key="hibernate.dialect">org.hibernate.dialect.MySQLDialect”
25.         </prop>
26.              <prop key="hibernate.hbm2ddl.auto">update</prop>
27.              <prop key="hibernate.show_sql">true</prop>
28.          </props>
29.       </property>
30.    </bean>
31.  <bean id="temp" class="org.springframework.orm.hibernate3.HibernateTemplate">
32. <property name="sessionFactory" ref="mysessionFactory"></property>
33.    </bean>
34.    <bean id="d" class="EmployeeDao">
35.    <property name="template" ref="temp"></property>
36.    </bean>
37.   </beans>
```

6. **InsertTest.java**

This class uses the EmployeeDao class object and calls its saveEmployee method by passing the object of Employee class.

```
1. import org.springframework.beans.factory.BeanFactory;
2. import org.springframework.beans.factory.xml.XmlBeanFactory;
3. import org.springframework.core.io.ClassPathResource;
4. import org.springframework.core.io.Resource;
5. public class InsertTest {
6. public static void main(String[] args) {
7.     Resource r=new ClassPathResource("applicationContext.xml");
8.     BeanFactory factory=new XmlBeanFactory(r);
9.     EmployeeDao dao=(EmployeeDao)factory.getBean("d");
10.    Employee e=new Employee();
11.    e.setId(114);
12.    e.setName("Mr Varun Dhavan");
13.    e.setSalary(50000);
14.    dao.saveEmployee(e);
15. } }
```

Spring Data JPA Tutorial

- Spring Data JPA API provides JpaTemplate class to integrate spring application with JPA.
- JPA (Java Persistent API) is the sun specification for persisting objects in the enterprise application. It is currently used as the replacement for complex entity beans.
- The implementation of JPA specification are provided by many vendors such as:
    - Hibernate
    - Toplink
    - iBatis
    - OpenJPA etc.

Advantage of Spring JpaTemplate

- You don't need to write the before and after code for persisting, updating, deleting or searching object such as creating Persistence instance, creating EntityManagerFactory instance, creating EntityTransaction instance, creating EntityManager instance, commiting EntityTransaction instance and closing EntityManager.
- In this example, we are going to use hibernate for the implementation of JPA.

Example of Spring and JPA Integration

Let's see the simple steps to integration spring application with JPA:

1. **create Account.java file**
2. **create Account.xml file**
3. **create AccountDao.java file**

4. **create persistence.xml file**
5. **create applicationContext.xml file**
6. **create AccountsClient.java file**

In this example, we are going to integrate the hibernate application with spring. Let's see the **directory structure** of jpa example with spring.

1. **Account.java**

```
1. public class Account {
2.     private int accountNumber;
3.     private String owner;
4.     private double balance;
5.         //no-arg and parameterized constructor  and getters-setters
6. }
```

2. Account.xml

This mapping file contains all the information of the persistent class.

```
1. <entity-mappings version="1.0"
2.         xmlns="http://java.sun.com/xml/ns/persistence/orm"
3.         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.         xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm
5.         http://java.sun.com/xml/ns/persistence/orm_1_0.xsd ">
6.       <entity class="Account">
7.         <table name="account100"></table>
8.         <attributes>
9.             <id name="accountNumber">
10.             <column name="accountnumber"/>
11.            </id>
12.            <basic name="owner">
13.               <column name="owner"/>
14.            </basic>
15.            <basic name="balance">
16.               <column name="balance"/>
17.            </basic>
18.        </attributes>
19.    </entity>
20. </entity-mappings>
```

3. AccountDao.java

```
1. import java.util.List;
2. import org.springframework.orm.jpa.JpaTemplate;
3. import org.springframework.transaction.annotation.Transactional;
4. @Transactional
```

```
5.  public class AccountsDao{
6.      JpaTemplate template;
7.      public void setTemplate(JpaTemplate template) {
8.          this.template = template;      }
9.      public void createAccount(int accountNumber,String owner,double balance){
10.         Account account = new Account(accountNumber,owner,balance);
11.         template.persist(account);   }
12.     public void updateBalance(int accountNumber,double newBalance){
13.         Account account = template.find(Account.class, accountNumber);
14.         if(account != null){
15.             account.setBalance(newBalance);       }
16.         template.merge(account);   }
17.     public void deleteAccount(int accountNumber){
18.         Account account = template.find(Account.class, accountNumber);
19.         if(account != null)
20.             template.remove(account);   }
21.     public List<Account> getAllAccounts(){
22.         List<Account> accounts =template.find("select acc from Account acc");
23.         return accounts;   }  }
```

4. persistence.xml

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <persistence xmlns="http://java.sun.com/xml/ns/persistence"
3.      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.      xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
5.      http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd" version="1.0">
6.       <persistence-unit name="ForAccountsDB">
7.          <mapping-file> Account.xml</mapping-file>
8.          <class>Account</class>
9.      </persistence-unit>
10. </persistence>
```

5. 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:tx="http://www.springframework.org/schema/tx"
5.      xsi:schemaLocation="http://www.springframework.org/schema/beans
6.      http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
7.      http://www.springframework.org/schema/tx
8.      http://www.springframework.org/schema/tx/spring-tx-3.0.xsd">
```

```
<tx:annotation-driven transaction-manager="jpaTxnManagerBean" proxy-target-class="true"/>
    <bean id="dataSourceBean" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
  <property name="driverClassName" value="com.jdbc.mysql.Driver"></property>
   <property name="url" value="jdbc:mysql://localhost:3306/TestDB"></property>
      <property name="username" value="root"></property>
      <property name="password" value="root"></property>
   </bean>
  <bean id="hbAdapterBean" class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">
  <property name="showSql" value="true"></property>
      <property name="generateDdl" value="true"></property>
 <property name="databasePlatform" value="org.hibernate.dialect.MySQLDialect">
 </property>  </bean>
  <bean id="emfBean" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
      <property name="dataSource" ref="dataSourceBean"></property>
      <property name="jpaVendorAdapter" ref="hbAdapterBean"></property>
    </bean>
    <bean id="jpaTemplateBean" class="org.springframework.orm.jpa.JpaTemplate">
     <property name="entityManagerFactory" ref="emfBean"></property>
    </bean>
     <bean id="accountsDaoBean" class="AccountsDao">
      <property name="template" ref="jpaTemplateBean"></property>
    </bean>
     <bean id="jpaTxnManagerBean" class="org.springframework.orm.jpa.JpaTransactionManager">
      <property name="entityManagerFactory" ref="emfBean"></property>
   </bean>
</beans>
```

6. Accountsclient.java

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

```
8.  AccountsDao accountsDao = context.getBean("accountsDaoBean",AccountsDao.cla
    ss);
9.  accountsDao.createAccount(15, "Jai Kumar", 41000);
10. accountsDao.createAccount(20, "Rishi ", 35000);
11. System.out.println("Accounts created");
12. }  }
```

Output

```
Hibernate: insert into account100 (balance, owner, accountnumber) values (?, ?, ?)
Hibernate: insert into account100 (balance, owner, accountnumber) values (?, ?, ?)
Accounts created
```

Spring Expression Language (SPEL)

- **SpEL** is an exression language supporting the features of querying and manipulating an object graph at runtime.
- There are many expression languages available such as JSP EL, OGNL, MVEL and JBoss EL.
- SpEL provides some additional features such as method invocation and string templating functionality.

SpEL API

The SpEL API provides many interfaces and classes. They are as follows:

- Expression interface
- SpelExpression class
- ExpressionParser interface
- SpelExpressionParser class
- EvaluationContext interface
- StandardEvaluationContext class

Hello SPEL Example

```
1.  import org.springframework.expression.Expression;
2.  import org.springframework.expression.ExpressionParser;
3.  import org.springframework.expression.spel.standard.SpelExpressionParser;
4.  public class Test {
5.  public static void main(String[] args) {
6.  ExpressionParser parser = new SpelExpressionParser();
7.  Expression exp = parser.parseExpression("'Hello SPEL'");
8.  String message = (String) exp.getValue();
9.  System.out.println(message);  //OR
10. //System.out.println(parser.parseExpression("'Hello SPEL'").getValue());
11. }  }
```

Other SPEL Example

- Let's see a lot of useful examples of SPEL. Here, we are assuming all the examples have been written inside the main() method.

Using concat() method with String

```
1. ExpressionParser parser = new SpelExpressionParser();
2. Expression exp = parser.parseExpression("'Welcome SPEL'.concat('!')");
3. String message = (String) exp.getValue();
4. System.out.println(message);
```

Converting String into byte array

```
1. Expression exp = parser.parseExpression("'Hello World'.bytes");
2. byte[] bytes = (byte[]) exp.getValue();
3. for(int i=0;i<bytes.length;i++){
4.     System.out.print(bytes[i]+" ");
5. }
```

Getting length after converting string into bytes

```
1. Expression exp = parser.parseExpression("'Hello World'.bytes.length");
2. int length = (Integer) exp.getValue();
3. System.out.println(length);
```

Converting String contents into uppercase letter

```
1. Expression exp = parser.parseExpression("new String('hello world').toUpperCase()");
2. String message = exp.getValue(String.class);
3. System.out.println(message);  //OR
4. System.out.println(parser.parseExpression("'hello world'.toUpperCase()").getValue());
```

Operators in SPEL

1. Examples of Operators in SPEL

We can use many operators in SpEL such as arithmetic, relational, logical etc. There are given a lot of examples of using different operators in SpEL.

Examples of using operators in SPEL

```
1. import org.springframework.expression.ExpressionParser;
2. import org.springframework.expression.spel.standard.SpelExpressionParser;
3. public class Test {
4. public static void main(String[] args) {
5. ExpressionParser parser = new SpelExpressionParser();
6.  //arithmetic operator
7. System.out.println(parser.parseExpression("'Welcome SPEL'+'!'").getValue());
```

```
8.  System.out.println(parser.parseExpression("10 * 10/2").getValue());
9.  System.out.println(parser.parseExpression("'Today is: '+ new java.util.Date()").
                                                                    getValue());
10.  //logical operator
11. System.out.println(parser.parseExpression("true and true").getValue());
12.  //Relational operator
13. System.out.println(parser.parseExpression("'sonoo'.length()==5").getValue());
14. }  }
```

Variable in SPEL

- In SpEL, we can store a value in the variable and use the variable in the method and call the method. To work on variable, we need to use **StandardEvaluationContext** class.

Example of Using variable in SPEL

Calculation.java

```
1.  public class Calculation {
2.  private int number;
3.  public int getNumber() {
4.      return number;  }
5.  public void setNumber(int number) {
6.      this.number = number;  }
7.  public int cube(){
8.      return number*number*number;  }
9.  }
```

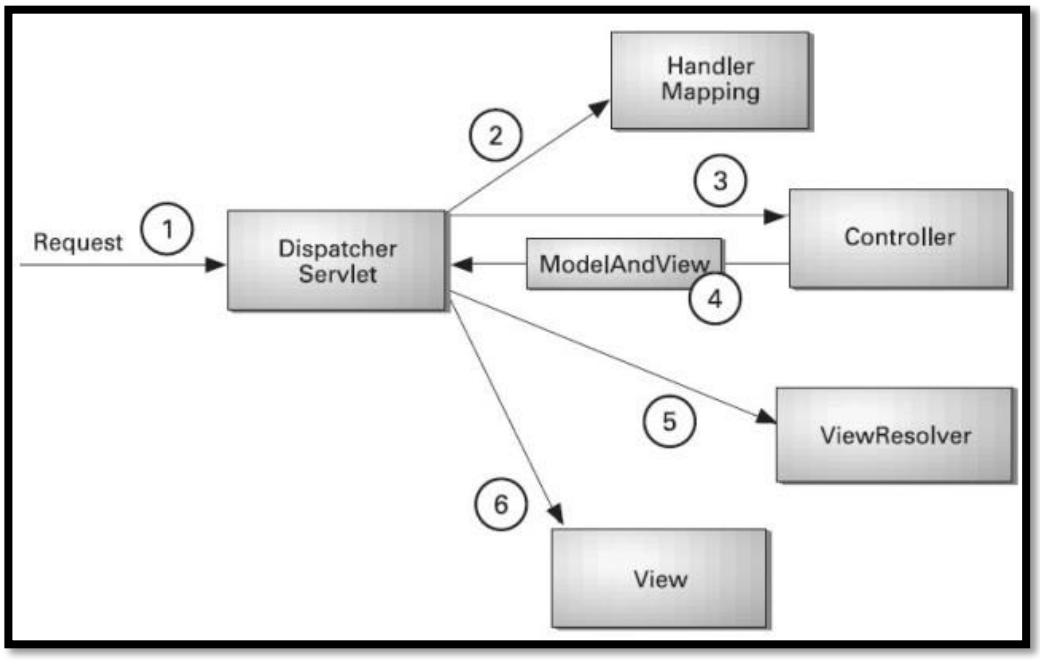
Test.java

```
1.  import org.springframework.expression.ExpressionParser;
2.  import org.springframework.expression.spel.standard.SpelExpressionParser;
3.  import org.springframework.expression.spel.support.StandardEvaluationContext;
4.  public class Test {
5.  public static void main(String[] args) {
6.  Calculation calculation=new Calculation();
7.  StandardEvaluationContext context=new StandardEvaluationContext(calculation);
8.  ExpressionParser parser = new SpelExpressionParser();
9.  parser.parseExpression("number").setValue(context,"5");
10. System.out.println(calculation.cube());
11. }  }
```

Spring MVC Tutorial

- **Spring MVC** tutorial provides an elegant solution to use MVC in spring framework by the help of DispatcherServlet.
- In Spring Web MVC, **DispatcherServlet** class works as the front controller. It is responsible to manage the flow of the spring mvc application.
- The **@Controller** annotation is used to mark the class as the controller in Spring 3.
- The **@RequestMapping** annotation is used to map the request url.
- It is applied on the method.

Flow of Spring Web MVC



- As displayed in the figure, all the incoming request is intercepted by the DispatcherServlet that works as the front controller.
- The DispatcherServlet gets entry of handler mapping from the xml file and forwards the request to the controller.
- The controller returns an object of ModelAndView.
- The DispatcherServlet checks the entry of view resolver in the xml file and invokes the specified view component.

Spring MVC Example

Let's see the simple example of spring 3 web MVC. There are given 7 steps for creating the spring MVC application. The steps are as follows:

1. **Create the request page (optional)**
2. **Create the controller class**
3. **Provide the entry of controller in the web.xml file**

4. **Define the bean in the xml file**
5. **Display the message in the JSP page**
6. **Load the spring core and mvc jar files**
7. **Start server and deploy the project**

Required Jar files

To run this example, you need to load:

- **Spring Core jar files**
- **Spring Web jar files**

1. Create the request page (optional)

This is the simple jsp page containing a link. It is optional page. You may direct invoke the action class instead.

index.jsp

```
1. <a href="hello.html">click</a>
```

2. Create the controller class

- To create the controller class, we are using two annotations @Controller and @RequestMapping.
- The **@Controller** annotation marks this class as Controller.
- The **@Requestmapping** annotation is used to map the class with the specified name.
- This class returns the instance of ModelAndView controller with the mapped name, message name and message value.
- The message value will be displayed in the jsp page.

HelloWorldController.java

```
1. import org.springframework.stereotype.Controller;
2. import org.springframework.web.bind.annotation.RequestMapping;
3. import org.springframework.web.servlet.ModelAndView;
4. @Controller
5. public class HelloWorldController {
6.     @RequestMapping("/hello")
7.     public ModelAndView helloWorld() {
8.         String message = "HELLO SPRING MVC HOW R U";
9.         return new ModelAndView("hellopage", "message", message);
10.    } }
```

3. Provide the entry of controller in the web.xml file

- In this xml file, we are specifying the servlet class DispatcherServlet that acts as the front controller in Spring Web MVC. All the incoming request for the html file will be forwarded to the DispatcherServlet.

web.xml

```
1.  <servlet>
2.     <servlet-name>spring</servlet-name>
```

```
3.      <servlet-class>org.springframework.web.servlet.DispatcherServlet
4.      </servlet-class>
5.      <load-on-startup>1</load-on-startup>
6.  </servlet>
7.  <servlet-mapping>
8.      <servlet-name>spring</servlet-name>
9.      <url-pattern>*.html</url-pattern>
10. </servlet-mapping>
```

4. Define the bean in the xml file

- This is the important configuration file where we need to specify the ViewResolver and View components.
- The **context:component-scan** element defines the base-package where DispatcherServlet will search the controller class.
- Here, the **InternalResourceViewResolver** class is used for the ViewResolver.
- The **prefix+string returned by controller+suffix** page will be invoked for the view component.
- This xml file should be located inside the WEB-INF directory.

spring-servlet.xml

```
1.      <context:component-scan  base-package=" " />
2.   <bean class="org.springframework.web.servlet.view.InternalResourceViewResol
     ver">
3.          <property name="prefix" value="/WEB-INF/jsp/" />
4.          <property name="suffix" value=".jsp" />
5.      </bean>
```

5. Display the message in the JSP page

- This is the simple JSP page, displaying the message returned by the Controller.
- It must be located inside the WEB-INF/jsp directory for this example only.

hellopage.jsp

```
1.  Message is: ${message}
```

Remoting in Spring Framework

- Spring framework makes the developement of remote-enabled services easy. It saves a lot of code by providing its own API.

Advantage of Spring Remoting

- The programmer needs to concentrate on business logic only not plumbing activities such as starting and stopping the server.
- Spring framework supports following remoting technologies:

- ✓ Remote Method Invocation (RMI)
- ✓ Spring's HTTP invoker
- ✓ Hessian
- ✓ Burlap
- ✓ JAX-RPC (J2EE 1.4 API)
- ✓ JAX-WS (Java EE 5 and Java EE 6 API)
- ✓ JMS

Remote Method Invocation (RMI)

- By the help of RmiServiceExporter and RmiProxyFactoryBean classes, spring framework supports RMI provided by Sun.

Spring's HTTP invoker

- Spring provides its own remoting service that allows serialization by HTTP.
- The classes used in HTTP Invoker are HttpInvokerServiceExporter and HttpInvokerProxyFactoryBean.

Hessian

- It also provides remoting service by using http protocol. It is provided by Coucho.
- The classes used in Hessian are HessianServiceExporter and HessianProxyFactoryBean.

Burlap

- It is same as Hessian but XML-based implementation provided by Coucho.
- The classes used in Burlap are BurlapServiceExporter and BurlapProxyFactoryBean.

JAX-RPC

- Spring provides remoting support for web services using JAX-RPC. It uses J2EE 1.4 API.

JAX-WS

- It is the successor of JAX-RPC. It uses Java EE 5 and Java EE 6 API.
- The classes used in JAX-WS are SimpleJaxWsServiceExporter and JaxWsPortProxyFactoryBean.

JMS

- Spring supports remoting service using JMS. The classes used in JMS are JmsInvokerServiceExporter and JmsInvokerProxyFactoryBean.

Spring and RMI Integration

- Spring RMI lets you expose your services through the RMI infrastructure.
- Spring provides an easy way to run RMI application by the help of
  - org.springframework.remoting.rmi.RmiProxyFactoryBean and
  - org.springframework.remoting.rmi.RmiServiceExporter classes.

RmiServiceExporter

It provides the exportation service for the rmi object. This service can be accessed via plain RMI or RmiProxyFactoryBean.

RmiProxyFactoryBean

It is the factory bean for Rmi Proxies. It exposes the proxied service that can be used as a bean reference.

Example of Spring and RMI Integration

Let's see the simple steps to integration spring application with RMI:

1. **Calculation.java**
2. **CalculationImpl.java**
3. **applicationContext.xml**
4. **client-beans.xml**
5. **Host.java**
6. **Client.java**

Required Jar files

To run this example, you need to load:

- **Spring Core jar files**
- **Spring Remoting jar files**
- **Spring AOP jar files**

Calculation.java

It is the simple interface containing one method cube.

```
1. public interface Calculation {
2. int cube(int number);  }
```

CalculationImpl.java

This class provides the implementation of Calculation interface.

```
1. public class CalculationImpl implements Calculation{
2.     @Override
3.     public int cube(int number) {
4.         return number*number*number;  }
5.  }
```

applicationContext.xml

In this xml file we are defining the bean for CalculationImpl class and **RmiServiceExporter** class. We need to provide values for the following properties of RmiServiceExporter class.

- ✓ service
- ✓ serviceInterface
- ✓ serviceName
- ✓ replaceExistingBinding
- ✓ registryPort

```
<bean id="cBean" class="CalculationImpl"></bean>
<bean class="org.springframework.remoting.rmi.RmiServiceExporter">
    <property name="service" ref="cBean"></property>
    <property name="serviceInterface" value="Calculation"></property>
    <property name="serviceName" value="CalculationService"></property>
    <property name="replaceExistingBinding" value="true"></property>
    <property name="registryPort" value="1099"></property>
</bean>
```

client-beans.xml

In this xml file, we are defining bean for **RmiProxyFactoryBean**. You need to define two properties of this class.

- ✓ serviceUrl
- ✓ serviceInterface

```
<bean id="cBean"class="org.springframework.remoting.rmi.RmiProxyFactoryBean">
<property name="serviceUrl" value="rmi://localhost:1099/CalculationService">
</property>
<property name="serviceInterface" value="Calculation"></property>
</bean>
```

Host.java

It is simply getting the instance of ApplicationContext. But you need to run this class first to run the example.

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
public class Host{
public static void main(String[] args){
ApplicationContext context = new ClassPathXmlApplicationContext("applicationCo
ntext.xml");
System.out.println("Waiting for requests");
} }
```

Client.java

This class gets the instance of Calculation and calls the method.

```
package com.javatpoint;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
 public class Client {
public static void main(String[] args)  {
ApplicationContext context = new ClassPathXmlApplicationContext("client-
beans.xml");
Calculation calculation = (Calculation)context.getBean("calculationBean");
System.out.println(calculation.cube(7));
```

```
9.  } }
```

Spring Remoting by HTTP Invoker

- Spring provides its own implementation of remoting service known as **HttpInvoker**. It can be used for http request than RMI and works well across the firewall.
- By the help of **HttpInvokerServiceExporter** and **HttpInvokerProxyFactoryBean** classes, we can implement the remoting service provided by Spring's Http Invokers.

Http Invoker and Other Remoting techniques

Http Invoker Vs RMI

- RMI uses JRMP protocol whereas Http Invokes uses HTTP protocol.
- Since enterprise applications mostly use http protocol, it is the better to use HTTP Invoker.
- RMI also has some security issues than HTTP Invoker. HTTP Invoker works well across firewalls.

Http Invoker Vs Hessian and Burlap

- HTTP Invoker is the part of Spring framework but Hessian and burlap are proprietary. All works well across firewall. Hessian and Burlap are portable to integrate with other languages such as .Net and PHP but HTTP Invoker cannot be.

Example of Spring HTTP Invoker

To create a simple spring's HTTP invoker application, you need to create following files.

1. **Calculation.java**
2. **CalculationImpl.java**
3. **web.xml**
4. **httpInvoker-servlet.xml**
5. **client-beans.xml**
6. **Client.java**

Calculation.java

It is the simple interface containing one method cube.

```
1. public interface Calculation {
2. int cube(int number);
3. }
```

CalculationImpl.java

This class provides the implementation of Calculation interface.

```
1. public class CalculationImpl implements Calculation{
2.     public int cube(int number) {
3.         return number*number*number;
4.     }
5. }
```

web.xml

In this xml file, we are defining DispatcherServlet as the front controller. If any request is followed by .http extension, it will be forwarded to DispatcherServlet.

```
1.      <servlet>
2.      <servlet-name>httpInvoker</servlet-name>
3.      <servlet-class>org.springframework.web.servlet.DispatcherServlet
4.      </servlet-class>
5.      <load-on-startup>1</load-on-startup>
6.  </servlet>
7.   <servlet-mapping>
8.      <servlet-name>httpInvoker</servlet-name>
9.      <url-pattern>*.http</url-pattern>
10. </servlet-mapping>
```

httpInvoker-servlet.xml

It must be created inside the WEB-INF folder. Its name must be servletname-servlet.xml. It defines bean for **CalculationImpl** and **HttpInvokerServiceExporter**.

```
1.  <bean id="calculationBean" class="CalculationImpl"></bean>
2.  <bean name="/Calculation.http"
3.  class="org.springframework.remoting.httpinvoker.HttpInvokerServiceExporter">
4.      <property name="service" ref="calculationBean"></property>
5.   <property name="serviceInterface" value="Calculation"></property>
6.  </bean>
```

client-beans.xml

In this xml file, we are defining bean for **HttpInvokerProxyFactoryBean**. You need to define two properties of this class.

- ✓ serviceUrl
- ✓ serviceInterface

```
1.  <bean id="calculationBean"
2.  class="org.springframework.remoting.httpinvoker.HttpInvokerProxyFactoryBean">
3.      <property name="serviceUrl"
4.          value="http://localhost:8888/httpinvoker/Calculation.http"></property>
5.   <property name="serviceInterface" value=" Calculation"></property>
6.  </bean>
7.  </beans>
```

Client.java

This class gets the instance of Calculation and calls the method.

```
1.  import org.springframework.context.ApplicationContext;
2.  import org.springframework.context.support.ClassPathXmlApplicationContext;
3.  public class Client {
```

```
4.  public static void main(String[] args){
5.   ApplicationContext context = new ClassPathXmlApplicationContext("client-
     beans.xml");
6.   Calculation calculation = (Calculation)context.getBean("calculationBean");
7.   System.out.println(calculation.cube(5));
8.  } }
```

Output 125

Web-based Client

In the example given above, we used console based client. We can also use web based client. You need to create 3 additional files. Here, we are using following files:

1. ClientInvoker.java
2. index.jsp
3. process.jsp

ClientInvoker.java

It defines only one method getCube() that returns cube of the given number

```
1. import org.springframework.context.ApplicationContext;
2. import org.springframework.context.support.ClassPathXmlApplicationContext;
3. public class ClientInvoker {
4.     public static int getCube(int number){
5.         ApplicationContext context = new ClassPathXmlApplicationContext("client-
     beans.xml");
6.         Calculation calculation = (Calculation)context.getBean("calculationBean");
7.         return calculation.cube(number);      }  }
```

index.jsp

```
1. <form action="process.jsp">
2. Enter Number:<input type="text" name="number"/>
3. <input type="submit" value="cube" />
4. </form>
```

process.jsp

```
1. <jsp:include page="index.jsp"></jsp:include>
2. <hr/>
3. <%@page import="ClientInvoker"%>
4.  <%
5. int number=Integer.parseInt(request.getParameter("number"));
6. out.print("cube of "+number+" is: "+ClientInvoker.getCube(number));
```

```
7. %>
```

Spring Remoting by Hessian

- By the help of **HessianServiceExporter** and **HessianProxyFactoryBean** classes, we can implement the remoting service provided by hessian.

Advantage of Hessian

Hessian works well across firewall. Hessian is portable to integrate with other languages such as PHP and .Net.

Example of Remoting by Hessian

You need to create following files for creating a simple hessian application:

1. **Calculation.java**
2. **CalculationImpl.java**
3. **web.xml**
4. **hessian-servlet.xml**
5. **client-beans.xml**
6. **Client.java**

Calculation.java

It is the simple interface containing one method cube.

```
1. public interface Calculation {
2. int cube(int number);  }
```

CalculationImpl.java

This class provides the implementation of Calculation interface.

```
1. public class CalculationImpl implements Calculation{
2.     public int cube(int number) {
3.         return number*number*number;
4.     } }
```

web.xml

In this xml file, we are defining DispatcherServlet as the front controller. If any request is followed by .http extension, it will be forwarded to DispatcherServlet.

```
1.  <servlet>
2.    <servlet-name>hessian</servlet-name>
3.    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-
    class>
4.    <load-on-startup>1</load-on-startup>
5.  </servlet>
6.   <servlet-mapping>
7.    <servlet-name>hessian</servlet-name>
8.    <url-pattern>*.http</url-pattern>
9.  </servlet-mapping>
```

hessian-servlet.xml

It must be created inside the WEB-INF folder. Its name must be servletname-servlet.xml. It defines bean for **CalculationImpl** and **HessianServiceExporter**.

```
1.  <bean id="calculationBean" class="CalculationImpl"></bean>
2.  <bean name="/Calculation.http"
3.  class="org.springframework.remoting.caucho.HessianServiceExporter">
4.    <property name="service" ref="calculationBean"></property>
5.    <property name="serviceInterface" value="Calculation">
6.  </property>
7.  </bean>
```

client-beans.xml

In this xml file, we are defining bean for **HessianProxyFactoryBean**. You need to define two properties of this class.

- ✓ serviceUrl
- ✓ serviceInterface

```
1.  <bean id="calculationBean"
2.  class="org.springframework.remoting.caucho.HessianProxyFactoryBean">
3.    <property name="serviceUrl"
4.        value="http://localhost:8888/hessian/Calculation.http"></property>
5.    <property name="serviceInterface" value="Calculation"></property>
6.  </bean>
```

Client.java

This class gets the instance of Calculation and calls the cube method.

```
1.  import org.springframework.context.ApplicationContext;
2.  import org.springframework.context.support.ClassPathXmlApplicationContext;
3.  public class Client {
4.   public static void main(String[] args){
```

```
5. ApplicationContext context = new ClassPathXmlApplicationContext("client-
   beans.xml");
6. Calculation calculation = (Calculation)context.getBean("calculationBean");
7.   System.out.println(calculation.cube(5));
8.  } }
```

Spring Remoting by Burlap Example

Both, Hessian and Burlap are provided by Coucho. Burlap is the xml-based alternative of Hessian.

By the help of **BurlapServiceExporter** and **BurlapProxyFactoryBean** classes, we can implement the remoting service provided by burlap.

Example of Burlap is same as Hessian, you need to change Hessian to Burlap only.

Example of Remoting by Burlap

You need to create following files for creating a simple burlap application:

1. **Calculation.java**
2. **CalculationImpl.java**
3. **web.xml**
4. **burlap-servlet.xml**
5. **client-beans.xml**
6. **Client.java**

Calculation.java

```
1. public interface Calculation {
2. int cube(int number);  }
```

CalculationImpl.java

```
1. public class CalculationImpl implements Calculation{
2.     public int cube(int number) {
3.         return number*number*number;      }  }
```

web.xml

```
1.     <servlet>
2.      <servlet-name>burlap</servlet-name>
3.      <servlet-class>org.springframework.web.servlet.DispatcherServlet
4.      </servlet-class>
5.      <load-on-startup>1</load-on-startup>
6.     </servlet>
7.  <servlet-mapping>
8.      <servlet-name>burlap</servlet-name>
9.      <url-pattern>*.http</url-pattern>
10. </servlet-mapping>
```

burlap-servlet.xml

It must be created inside the WEB-INF folder. Its name must be servletname-servlet.xml. It defines bean for **CalculationImpl** and **BurlapServiceExporter**.

```
1. <bean id="calculationBean" class="CalculationImpl"></bean>
2. <bean name="/Calculation.http"
3. class="org.springframework.remoting.caucho.BurlapServiceExporter">
4.     <property name="service" ref="calculationBean"></property>
5.     <property name="serviceInterface" value="Calculation"></property>
6. </bean>
```

client-beans.xml

In this xml file, we are defining bean for **BurlapProxyFactoryBean**. You need to define two properties of this class.

- ✓ serviceUrl
- ✓ serviceInterface

```
1. <bean id="calculationBean"
2. class="org.springframework.remoting.caucho.BurlapProxyFactoryBean">
3.     <property name="serviceUrl"
4.         value="http://localhost:8888/burlap/Calculation.http"></property>
5.     <property name="serviceInterface" value="Calculation"></property>
6. </bean>
```

Client.java

```
1. import org.springframework.context.ApplicationContext;
2. import org.springframework.context.support.ClassPathXmlApplicationContext;
3. public class Client {
4.  public static void main(String[] args){
5.   ApplicationContext context = new ClassPathXmlApplicationContext("client-
   beans.xml");
6.   Calculation calculation = (Calculation)context.getBean("calculationBean");
7.   System.out.println(calculation.cube(3));
8.  } }
```

Spring and JMS Integration

- To integrate spring with JMS, you need to create two applications.
  1. JMS Receiver Application
  2. JMS Sender Application
- To create JMS application using spring, we are using **Active MQ Server** of Apache to create the Queue.

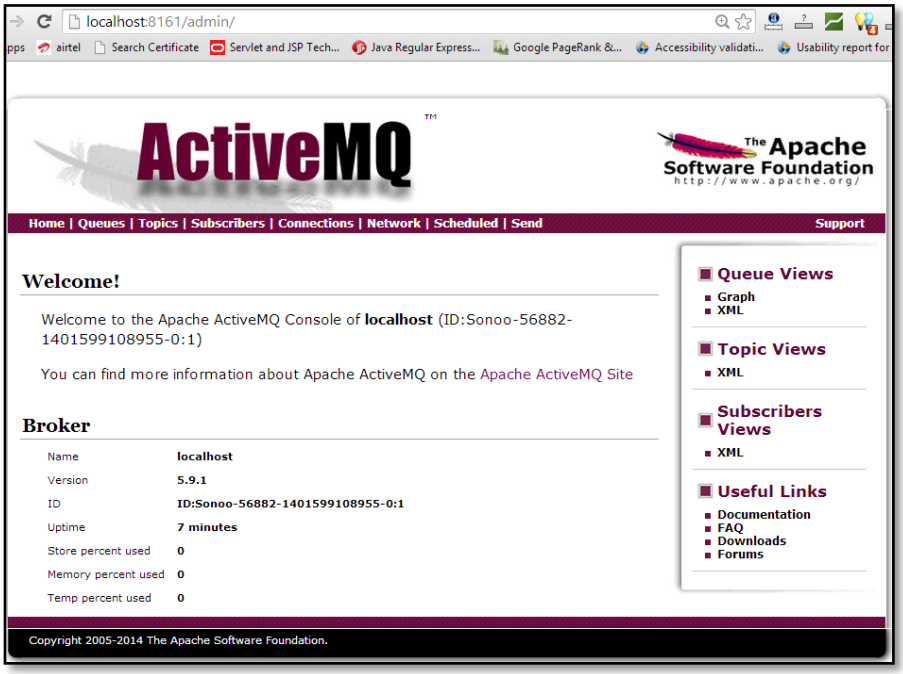
Let's see the simple steps to integration spring application with JMS:

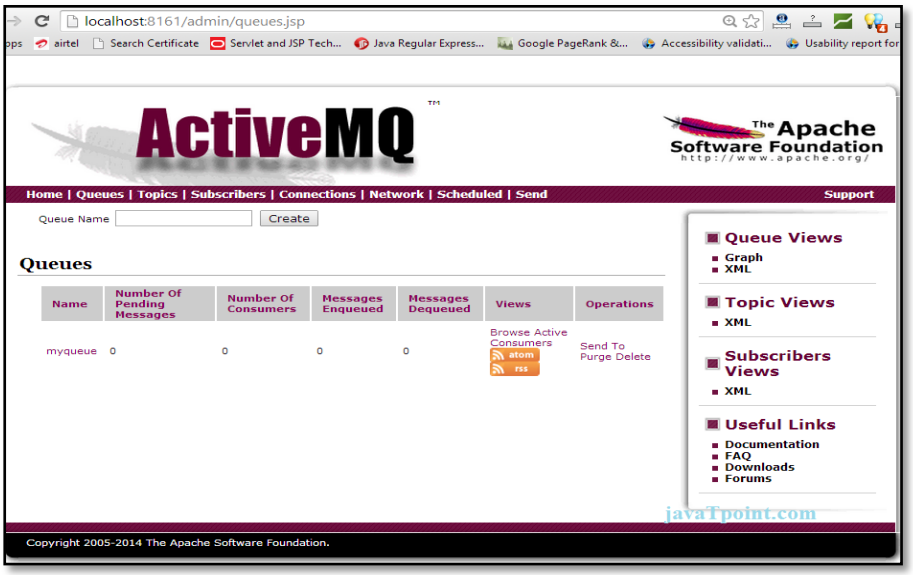
Required Jar Files

- You need to add **spring core**, **spring misc**,**spring aop**, **spring j2ee** and **spring persistence core** jar files.
- Add **activemqall5.9.jar** file located inside the activemq directory.

Create a queue in ActiveMQ Server

- Download the Active MQ Server
- Double Click on the **activemq.bat** file located inside apache-activemq-5.9.1-bin\apache-activemq-5.9.1\bin\win64 or win32 directory.
- Now activemq server console will open.
- Access the admin console of activemq server by **http://localhost:8161/admin/** url.
- Now, click on the **Queues** link, write **myqueue** in the textfield and click on the create button.





JMS Receiver Application

Let's see the simple steps to integration spring application with JMS:

1. **MyMessageListener.java**
2. **TestListener.java**
3. **applicationContext.xml**

MyMessageListener.java

```
1. import javax.jms.*;
2. public class MyMessageListener implements MessageListener{
3.     @Override
4.     public void onMessage(Message m) {
5.         TextMessage message=(TextMessage)m;
6.         try{
7.             System.out.println(message.getText());
8.         }catch (Exception e) {e.printStackTrace();  }
9.     }  }
```

TestListener.java

```
1. import org.springframework.context.support.GenericXmlApplicationContext;
2. public class TestListener {
3. public static void main(String[] args) {
4.     GenericXmlApplicationContext ctx=new GenericXmlApplicationContext();
5.     ctx.load("classpath:applicationContext.xml");
6.     ctx.refresh();
7.    while(true){}
8. }  }
```

applicationContext.xml

```
1. <bean id="cFactory" class="org.apache.activemq.ActiveMQConnectionFactory"
2.    p:brokerURL="tcp://localhost:61616" />
3.   <bean id="listener" class="MyMessageListener"></bean>
4.   <jms:listener-container container-type="default" connection-factory =
   "cFactory"  acknowledge="auto">
5. <jms:listener destination="myqueue" ref="listener" method="onMessage">
6. </jms:listener>
7. </jms:listener-container>
```

JMS Sender Application

Let's see the files to create the JMS Sender application:

1. **MyMessageSender.java**
2. **TestJmsSender.java**
3. **applicationContext.xml**

MyMessageListener.java

```
1. import javax.jms.*;
2. import org.springframework.beans.factory.annotation.Autowired;
3. import org.springframework.jms.core.JmsTemplate;
4. import org.springframework.jms.core.MessageCreator;
5. import org.springframework.stereotype.Component;
6. @Component("messageSender")
7. public class MyMessageSender {
8. @Autowired
9. private JmsTemplate jmsTemplate;
10. public void sendMessage(final String message){
11.     jmsTemplate.send(new MessageCreator(){
12.       @Override
13.        public Message createMessage(Session session) throws JMSException {
14.            return session.createTextMessage(message);
15.        }
16.     });
17. }  }
```

TestJmsSender.java

```
1. import org.springframework.context.support.GenericXmlApplicationContext;
2. public class TestJmsSender {
3. public static void main(String[] args) {
4.     GenericXmlApplicationContext ctx=new GenericXmlApplicationContext();
5.     ctx.load("classpath:applicationContext.xml");
6.     ctx.refresh();
7.     MyMessageSender sender=ctx.getBean("messageSender",MyMessageSender.class
   );
8.     sender.sendMessage("hello jms3");
9.  }  }
```

3) applicationContext.xml

```
1. <bean id="cFactory" class="org.apache.activemq.ActiveMQConnectionFactory"
2. p:brokerURL="tcp://localhost:61616" />
3.   <bean id="jmsTemplate" class="org.springframework.jms.core.JmsTemplate">
4. <constructor-arg name="connectionFactory" ref="cFactory"></constructor-arg>
5. <property name="defaultDestinationName" value="myqueue"></property>
6. </bean>
7.  <context:component-scan base-package=" "></context:component-scan>
```

Unit 3

OXM Framework: Spring and JAXB Integration

- JAXB is an acronym for **Java Architecture for XML Binding**.
- It allows java developers to map Java class to XML representation.
- JAXB can be used to marshal java objects into XML and vice-versa.
- It is an OXM (Object XML Mapping) or O/M framework.

Advantage of JAXB

No need to create or use a SAX or DOM parser and write callback methods.

Example of Spring and JAXB Integration

You need to create following files for marshalling java object into XML using Spring with JAXB:

1. **Employee.java**
2. **applicationContext.xml**
3. **Client.java**

Required Jar files

To run this example, you need to load:

- **Spring Core jar files**
- **Spring Web jar files**

Employee.java

If defines three properties id, name and salary. We have used following annotations in this class:

- **@XmlRootElement** It specifies the root element for the xml file.
- **@XmlAttribute** It specifies attribute for the property.
- **@XmlElement** It specifies the element.

```
1. import javax.xml.bind.annotation.*;
2. @XmlRootElement(name="employee")
3. public class Employee {
4. private int id;
5. private String name;
6. private float salary;
7. @XmlAttribute(name="id")
8. public int getId() {
9.     return id;  }
10. public void setId(int id) {
11.     this.id = id;  }
12. @XmlElement(name="name")
13. public String getName() {
```

```
14.     return name;  }
15. public void setName(String name) {
16.     this.name = name;  }
17. @XmlElement(name="salary")
18. public float getSalary() {
19.     return salary;  }
20. public void setSalary(float salary) {
21.     this.salary = salary;  }  }
```

applicationContext.xml

```
1.          <oxm:jaxb2-marshaller id="jaxbMarshallerBean">
2.              <oxm:class-to-be-bound name="Employee"/>
3.          </oxm:jaxb2-marshaller>
```

Client.java

It gets the instance of Marshaller from the applicationContext.xml file and calls the marshal method.

```
1.  import java.io.*;
2.  import javax.xml.transform.stream.StreamResult;
3.  import org.springframework.context.ApplicationContext;
4.  import org.springframework.context.support.ClassPathXmlApplicationContext;
5.  import org.springframework.oxm.Marshaller;
6.  public class Client{
7.   public static void main(String[] args)throws IOException{
8.    ApplicationContext context = new ClassPathXmlApplicationContext("applicationC
    ontext.xml");
9.    Marshaller marshaller = (Marshaller)context.getBean("jaxbMarshallerBean");
10.  Employee employee=new Employee();
11.  employee.setId(101);
12.  employee.setName("Bijal Parekh");
13.  employee.setSalary(100000);
14. marshaller.marshal(employee, new StreamResult(new FileWriter("employee.xml")));
15. System.out.println("XML Created Sucessfully");
16.  }  }
```

Output of the example

employee.xml

```
1.  <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2.  <employee id="101">
3.  <name>Bijal Parekh</name>
4.  <salary>100000.0</salary>
5.  </employee>
```

Spring with Xstream Example

- **Xstream** is a library to serialize objects to xml and vice-versa without requirement of any mapping file. Notice that castor requires an mapping file.
- **XStreamMarshaller** class provides facility to marshal objects into xml and vice-versa.

Example of Spring and Xstream Integration

You need to create following files for marshalling java object into XML using Spring with Xstream:

1. **Employee.java**
2. **applicationContext.xml**
3. **Client.java**

Required Jar files

To run this example, you need to load:

- **Spring Core jar files**
- **Spring Web jar files**
- **xstream-1.3.jar**

Employee.java

If defines three properties id, name and salary with setters and getters.

```
1. public class Employee {
2. private int id;
3. private String name;
4. private float salary;
5. public int getId() {
6.     return id;  }
7. public void setId(int id) {
8.     this.id = id;  }
9. public String getName() {
10.    return name;  }
11. public void setName(String name) {
12.    this.name = name;  }
13. public float getSalary() {
14.    return salary;  }
15. public void setSalary(float salary) {
16.    this.salary = salary;  }
17. }
```

applicationContext.xml

```
1. <bean id="xstreamMarshallerBean" class="org.springframework.oxm.xstream.XStre
   amMarshaller">
2.  <property name="annotatedClasses" value="Employee"></property>
3. </bean>
```

Client.java

```
1. import java.io.*;
2. import javax.xml.transform.stream.StreamResult;
3. import org.springframework.context.ApplicationContext;
4. import org.springframework.context.support.ClassPathXmlApplicationContext;
5. import org.springframework.oxm.Marshaller;
6. public class Client{
7.  public static void main(String[] args)throws IOException{
8.   ApplicationContext context = new ClassPathXmlApplicationContext("applicationC
     ontext.xml");
9.   Marshaller marshaller = (Marshaller)context.getBean("xstreamMarshallerBean");
10.  Employee employee=new Employee();
11.  employee.setId(101);
12.  employee.setName("Bijal Parekh");
13.  employee.setSalary(100000);
14. marshaller.marshal(employee, new StreamResult(new FileWriter("employee.xml")));
15.  System.out.println("XML Created Sucessfully");
16. } }
```

Output of the example

employee.xml

```
< Employee>
<id>101</id>
<name>Bijal Parekh</name>
<salary>100000.0</salary>
</ Employee>
```

Spring with Castor Example

- By the help of **CastorMarshaller** class, we can marshal the java objects into xml and vice-versa using castor.
- It is the implementation class for Marshaller and Unmarshaller interfaces. It doesn't require any further configuration bydefault.

Example of Spring and Castor Integration

You need to create following files for marshalling java object into XML using Spring with Castor:

1. **Employee.java**
2. **applicationContext.xml**
3. **mapping.xml**
4. **Client.java**

Required Jar files

To run this example, you need to load:

- **Spring Core jar files**
- **Spring Web jar files**
- **castor-1.3.jar**
- **castor-1.3-core.jar**

Employee.java

If defines three properties id, name and salary with setters and getters.

```
1. public class Employee {
2. private int id;
3. private String name;
4. private float salary;
5. public int getId() {
6.     return id;  }
7. public void setId(int id) {
8.     this.id = id;  }
9. public String getName() {
10.    return name;  }
11. public void setName(String name) {
12.    this.name = name;  }
13. public float getSalary() {
14.    return salary;  }
15. public void setSalary(float salary) {
16.    this.salary = salary;  }
17. }
```

applicationContext.xml

```
1. <bean id="castorMarshallerBean" class="org.springframework.oxm.castor.CastorMar
   shaller">
2.         <property name="targetClass" value="Employee"></property>
3.         <property name="mappingLocation" value="mapping.xml"></property>
4.     </bean>
```

mapping.xml

```
1.  <mapping>
2.     <class name="Employee" auto-complete="true" >
3.         <map-to xml="Employee" ns-uri=" " ns-prefix="dp"/>
4.         <field name="id" type="integer">
5.             <bind-xml name="id" node="attribute"></bind-xml>
6.         </field>
7.         <field name="name">
8.             <bind-xml name="name"></bind-xml>
9.         </field>
10.        <field name="salary">
```

```
11.            <bind-xml name="salary" type="float"></bind-xml>
12.         </field>
13.     </class>
14.  </mapping>
```

Client.java

```
1.  import java.io.*;
2.  import javax.xml.transform.stream.StreamResult;
3.  import org.springframework.context.ApplicationContext;
4.  import org.springframework.context.support.ClassPathXmlApplicationContext;
5.  import org.springframework.oxm.Marshaller;
6.  public class Client{
7.   public static void main(String[] args)throws IOException{
8.    ApplicationContext context = new ClassPathXmlApplicationContext("applicationC
    ontext.xml");
9.    Marshaller marshaller = (Marshaller)context.getBean("castorMarshallerBean");
10.  Employee employee=new Employee();
11.  employee.setId(101);
12.  employee.setName("Bijal Parekh");
13.  employee.setSalary(100000);
14. marshaller.marshal(employee, new StreamResult(new FileWriter("employee.xml")));
15.  System.out.println("XML Created Sucessfully");
16. } }
```

Output of the example

employee.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<dp:Employee xmlns:dp=" " id="101">
<dp:name>Bijal Parekh</dp:name>
<dp:salary>100000.0</dp:salary>
</dp:Employee>
```

Spring and Struts 2 Integration

- Spring framework provides an easy way to manage the dependency. It can be easily integrated with struts 2 framework.
- The **ContextLoaderListener** class is used to communicate spring application with struts 2. It must be specified in the web.xml file.

You need to follow following steps:

1. Create struts2 application and add spring jar files.
2. In **web.xml** file, define ContextLoaderListener class.
3. In **struts.xml** file, define bean name for the action class.

4. In **applicationContext.xml** file, create the bean. Its class name should be action class name e.g. com.javatpoint.Login and id should match with the action class of struts.xml file (e.g. login).
5. In the **action class**, define extra property e.g. message.

Example of Spring and Struts 2 Integration

You need to create following files for simple spring and struts 2 application:

1. **index.jsp**
2. **web.xml**
3. **struts.xml**
4. **applicationContext.xml**
5. **Login.java**
6. **welcome.jsp**
7. **error.jsp**

index.jsp

```
<%@ taglib uri="/struts-tags" prefix="s"%>
<s:form action="login">
<s:textfield name="userName" label="UserName"></s:textfield>
<s:submit></s:submit>
</s:form>
```

web.xml

```
 <welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
  </welcome-file-list>
  <filter>
    <filter-name>struts2</filter-name>
    <filter-class>
       org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter
    </filter-class>
  </filter>
  <listener>
<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
  <filter-mapping>
    <filter-name>struts2</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>
```

struts.xml

```
<struts>
<package name="abc" extends="struts-default">
```

```
3. <action name="login" class="login">
4. <result name="success">welcome.jsp</result>
5. </action>
6. </package>
7. </struts>
```

applicationContext.xml

```
1. <bean id="login" class="Login">
2. <property name="message" value="Welcome Spring"></property>
3. </bean>
```

Login.java

```
1. public class Login {
2. private String userName,message;
3. public String getMessage() {
4.     return message;  }
5. public void setMessage(String message) {
6.     this.message = message;  }
7. public String getUserName() {
8.     return userName;  }
9. public void setUserName(String userName) {
10.    this.userName = userName;  }
11. public String execute(){
12.    return "success";  }
13. }
```

welcome.jsp

It prints values of userName and message properties.

```
1. <%@ taglib uri="/struts-tags" prefix="s"%>
2. Welcome, <s:property value="userName"/><br/>
3. ${message}
```

error.jsp

It is the error page. But it is not required in this example because we are not defining any logic in the execute method of action class.

Login Example with Spring and Struts 2 Integration

In the previous example, we have simply integrated the spring application with struts 2. Now let's develop a login application with struts 2 and spring frameworks.

Example of Login application Spring and Struts2 Integration

You need to create following files :

1. **index.jsp**
2. **web.xml**

3. **struts.xml**
4. **applicationContext.xml**
5. **Login.java**
6. **login_success.jsp**
7. **login_error.jsp**

index.jsp

```
<%@ taglib uri="/struts-tags" prefix="s"%>
<s:form action="login">
<s:textfield name="name" label="Username"></s:textfield>
<s:password name="password" label="Password"></s:password>
<s:submit value="login"></s:submit>
</s:form>
```

web.xml

```
 <welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
  </welcome-file-list>
  <filter>
    <filter-name>struts2</filter-name>
    <filter-class>
       org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter
    </filter-class>
  </filter>
<listener>
<listener-class>org.springframework.web.context.ContextLoaderListener
 </listener-class>
</listener>
  <filter-mapping>
    <filter-name>struts2</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>
  </web-app>
```

struts.xml

```
<struts>
<package name="abc" extends="struts-default">
<action name="login" class="login">
<result name="success">login_success.jsp</result>
<result name="error">login_error.jsp</result>
</action>
  </package>  </struts>
```

applicationContext.xml

```
<bean id="login" class=" Login">
<property name="successmessage" value="You are successfully logged in!">
</property>
<property name="errormessage" value="Sorry, username or password error!">
</property>
</bean>
```

Login.java

```
public class Login {
private String name,password,successmessage,errormessage;
//setters and getters
  public String execute(){
    if(password.equals("admin")){
       return "success";      }
    else{
       return "error";      }
} }
```

login_success.jsp

It prints values of userName and message properties.

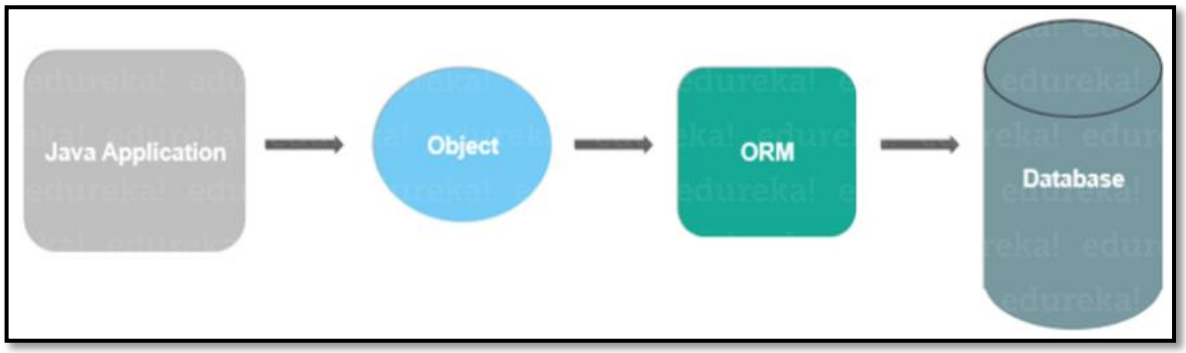
```
<%@ taglib uri="/struts-tags" prefix="s"%>
${successmessage}
<br/>
Welcome, <s:property value="name"/><br/>
```

login_error.jsp

```
${errormessage}
<jsp:include page="index.jsp"></jsp:include>
```

Hibernate

- Hibernate is a framework in Java which comes with an abstraction layer and handles the implementations internally. The implementations include tasks like writing a query for **CRUD** operations or establishing a connection with the databases, etc.
- A framework is basically a software that provides abstraction on multiple technologies like **JDBC, servlet**, etc.
- Hibernate develops persistence logic, which stores and processes the data for longer use. It is lightweight and an ORM tool, and most importantly open-source which gives it an edge over other frameworks.



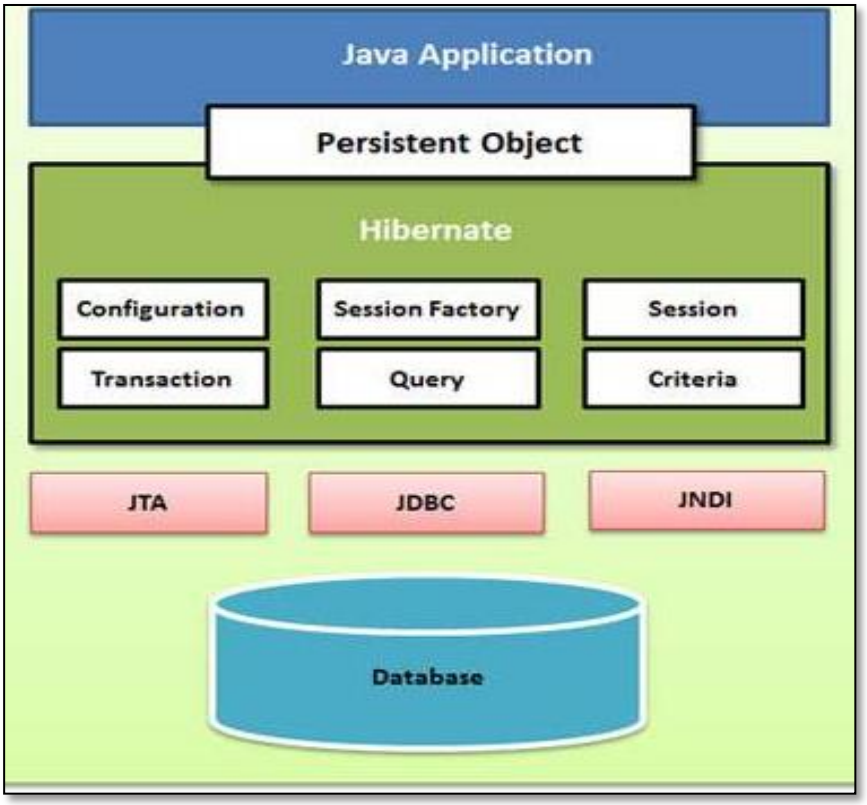
- Being an open-source framework, it is available for everyone without any cost.
- The source code can be found on the internet for hibernate which also allows modifications as well.
- The **advantage** of being a lightweight framework can be seen considerably smaller package for installation.
- The efficiency increases with not using **any container** for execution.

What is ORM?

- ORM – Object Relational Mapping
- Provides mapping using Java Objects
- Class to Table and Objects to Rows interactions
- ORM Tools:
  - Hibernate
  - TopLink (Oracle)
  - iBATIS (Persistance Framework)
  - JPA

Hibernate Architecture

- Hibernate uses various existing Java APIs, like JDBC, Java Transaction API(JTA), and Java Naming and Directory Interface (JNDI).
- JDBC provides a basic level of abstraction of functionality common to relational databases, allowing almost any database with a JDBC driver to be supported by Hibernate.
- JNDI and JTA allow Hibernate to be integrated with J2EE application servers.
- Following is Hibernate Application Architecture.



Configuration Object

- The Configuration object is the first Hibernate object you create in any Hibernate application.
- It is usually created only once during application initialization.
- It represents a configuration or properties file required by the Hibernate.
- There are two key componets of this object −
    - **Database Connection** − This is handled through one or more configuration files supported by Hibernate. These files are **hibernate.properties** and **hibernate.cfg.xml.**
    - **Class Mapping Setup** − This component creates the connection between the Java classes and database tables.

SessionFactory Object

- Configuration object is used to create a SessionFactory object.
- The SessionFactory is a **thread safe object** and used by all the threads of an application.
- The SessionFactory is a heavyweight object; it is usually created during application start up and kept for later use.
- You would need one SessionFactory object per database using a separate configuration file. So, if you are using multiple databases, then you would have to create multiple SessionFactory objects.

Session Object

- A Session is used to get a **physical connection** with a database.
- The Session object is **lightweight** and designed to be instantiated each time an interaction is needed with the database.
- Persistent objects are saved and retrieved through a Session object.
- The session objects should not be kept open for a long time because they are **not usually thread safe** and they should be created and destroyed them as needed.

Transaction Object

- A Transaction represents a unit of work with the database and most of the RDBMS supports transaction functionality.
- Transactions in Hibernate are handled by an underlying transaction manager and transaction
- This is an **optional object** and Hibernate applications may choose not to use this interface.

Query Object

- Query objects use SQL or Hibernate Query Language (HQL) string to retrieve data from the database and create objects.
- A Query instance is used to bind **query parameters**, limit the number of results returned by the query, and finally to execute the query.

Criteria Object

- Criteria objects are used to create and execute object oriented criteria queries to retrieve objects.

Hibernate Example using XML in Eclipse

- Here, we are going to create a simple example of hibernate application using eclipse IDE.
- For creating the first hibernate application in Eclipse IDE, we need to follow the following steps:
  1. Create the java project
  2. Add jar files for hibernate
  3. Create the Persistent class
  4. Create the mapping file for Persistent class
  5. Create the Configuration file
  6. Create the class that retrieves or stores the persistent object
  7. Run the application

Create the java project

- Create the java project by **File Menu** - **New** - **project** - **java project** . Now specify the project name e.g. firsthb then **next** - **finish** .

Add jar files for hibernate

- To add the jar files **Right click on your project** - **Build path** - **Add external archives**.

Create the Persistent class

- To create the persistent class, Right click on **src** - **New** - **Class** - specify the class with package name - **finish** .

Employee.java

```
1. public class Employee {
2. private int id;
3. private String firstName,lastName;
4. public int getId() {
5. return id;  }
6. public void setId(int id) {
7. this.id = id;  }
8. public String getFirstName() {
9. return firstName;  }
10. public void setFirstName(String firstName) {
11. this.firstName = firstName;  }
12. public String getLastName() {
13. return lastName;  }
14. public void setLastName(String lastName) {
15. this.lastName = lastName;
16. }
17. }
```

Create the mapping file for Persistent class

- To create the mapping file, Right click on **src** - **new** - **file** - specify the file name (e.g. employee.hbm.xml) - **ok**. It must be outside the package.

employee.hbm.xml

```
1.  <?xml version='1.0' encoding='UTF-8'?>
2.  <!DOCTYPE hibernate-mapping PUBLIC
3.   "-//Hibernate/Hibernate Mapping DTD 5.3//EN"
4.   "http://hibernate.sourceforge.net/hibernate-mapping-5.3.dtd">
5.     <hibernate-mapping>
6.    <class name="Employee" table="emp">
7.      <id name="id">
8.       <generator class="assigned"></generator>
9.      </id>
10.      <property name="firstName"></property>
11.     <property name="lastName"></property>
12.   </class>
13.  </hibernate-mapping>
```

Create the Configuration file

- The configuration file contains all the informations for the database such as connection_url, driver_class, username, password etc.
- The hbm2ddl.auto property is used to create the table in the database automatically.
- To create the configuration file, right click on src - new - file.
- Now specify the configuration file name e.g. hibernate.cfg.xml.

hibernate.cfg.xml

```
1.  <?xml version='1.0' encoding='UTF-8'?>
2.  <!DOCTYPE hibernate-configuration PUBLIC
3.            "-//Hibernate/Hibernate Configuration DTD 5.3//EN"
4.            "http://hibernate.sourceforge.net/hibernate-configuration-5.3.dtd">
5.  <hibernate-configuration>
6.     <session-factory>
7.          <property name="hbm2ddl.auto">update</property>
8.          <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
9.        <property name="connection.url">jdbc:mysql://localhost:3306/Test</property>
10.      <property name="connection.username">root</property>
11.       <property name="connection.password">root</property>
12.      <property name="connection.driver_class"> com.mysql.jdbc.Driver</property>
13.     <mapping resource="employee.hbm.xml"/>
14.     </session-factory>
15. </hibernate-configuration>
```

Create the class that exceutes the persistent object and Run the application

- In this class, we are simply storing the employee object to the database.

```
1. import org.hibernate.Session;
2. import org.hibernate.SessionFactory;
3. import org.hibernate.Transaction;
4. import org.hibernate.boot.Metadata;
5. import org.hibernate.boot.MetadataSources;
6. import org.hibernate.boot.registry.StandardServiceRegistry;
7. import org.hibernate.boot.registry.StandardServiceRegistryBuilder;
8. public class StoreData {
9.   public static void main( String[] args )
10.   {
11.     StandardServiceRegistry ssr = new StandardServiceRegistryBuilder().configure("
    hibernate.cfg.xml").build();
12.      Metadata meta = new MetadataSources(ssr).getMetadataBuilder().build();
13.      SessionFactory factory = meta.getSessionFactoryBuilder().build();
14.      Session session = factory.openSession();
15.      Transaction t = session.beginTransaction();
16.      Employee e1=new Employee();
17.          e1.setId(1);
18.          e1.setFirstName("Mr. Suraj");
19.          e1.setLastName("Modi");
20.     session.save(e1);
21.      t.commit();
22.      System.out.println("successfully saved");
23.       factory.close();
24.       session.close();
25.   }
26. }
```

Hibernate with Annotation with Example

- The hibernate application can be created with annotation. There are many annotations that can be used to create hibernate application such as @Entity, @Id, @Table etc.
- Hibernate Annotations are based on the JPA 2 specification and supports all the features.
- All the JPA annotations are defined in the **javax.persistence** package.
- The core **advantage of using** hibernate annotation is that you don't need to create mapping (hbm) file.

Employee.java

```
1. import javax.persistence.*;
2. @Entity
3. @Table(name= "emp")
4. public class Employee {
5. @Id
6. private int id;
7. private String firstName,lastName;
8. public int getId() {return id;}
9. public void setId(int id) { this.id = id; }
10. public String getFirstName() { return firstName;}
11. public void setFirstName(String firstName) { this.firstName = firstName; }
12. public String getLastName() { return lastName;}
13. public void setLastName(String lastName) {this.lastName = lastName; }
14. }
```

Hibernate.cfg.xml

```
1. <session-factory>
2. <property name="hbm2ddl.auto">update</property>
3. <property name="hibernate.dialect"> org.hibernate.dialect.MySQLDialect
   </property>
4. <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver
   </property>
5. <property name="hibernate.connection.url">jdbc:mysql://localhost:3306/test
   </property>
6. <property name="hibernate.connection.username">root</property>
7. <property name="hibernate.connection.password">root</property>
8. <mapping class="Employee"/>
9. </session-factory>
```

Test.java

```
1. import org.hibernate.*;
2. import org.hibernate.cfg.*;
3. public class Test {
4. public static void main(String[] args) {
5. Session session=new
        AnnotationConfiguration().configure().buildSessionFactory().openSession();
6. Transaction t=session.beginTransaction();
7. Employee e1=new Employee();
8.   e1.setId(1001);
9.   e1.setFirstName("Ever");
10.   e1.setLastName("Positive");
11. Employee e2=new Employee();
```

```
12.    e2.setId(1002);
13.    e2.setFirstName("Never");
14.    e2.setLastName("Negative");
15. session.persist(e1);
16. session.persist(e2);
17. t.commit();
18. session.close();
19. System.out.println("successfully saved");
20. }
21. }
```

Web Application with Hibernate (using XML)

- Here, we are going to create a web application with hibernate.
- For creating the web application, we are using JSP for presentation logic, Bean class for representing data and DAO class for database codes.
- As we create the simple application in hibernate, we don't need to perform any extra operations in hibernate for creating web application.
- In such case, we are getting the value from the user using the JSP file.

index.jsp

- This page gets input from the user and sends it to the register.jsp file using post method.

```
1. <form action="register.jsp" method="post">
2. Name:<input type="text" name="name"/><br><br/>
3. Password:<input type="password" name="password"/><br><br/>
4. Email ID:<input type="text" name="email"/><br><br/>
5. <input type="submit" value="register"/>"
6.   </form>
```

register.jsp

- This file gets all request parameters and stores this information into an object of User class. Further, it calls the register method of UserDao class passing the User class object.

```
1. <%@page import="UserDao"%>
2. <jsp:useBean id="obj" class="User">  </jsp:useBean>
3. <jsp:setProperty property="*" name="obj"/>
4.    <%   int i=UserDao.register(obj);
5.            if(i>0)
6.              out.print("You are successfully registered");
7. %>
```

User.java

- It is the simple bean class representing the Persistent class in hibernate.

```
1. public class User {
2. private int id;
3. private String name,password,email;
4.   //getters and setters
5. }
```

user.hbm.xml

```
1.  <hibernate-mapping>
2.  <class name="User" table="user">
3.  <id name="id">
4.  <generator class="increment"></generator>
5.  </id>
6.  <property name="name"></property>
7.  <property name="password"></property>
8.  <property name="email"></property>
9.  </class>
10. </hibernate-mapping>
```

UserDao.java

- A Dao class, containing method to store the instance of User class.

```
1.  import org.hibernate.*;
2.  import org.hibernate.*;
3.  import org.hibernate.boot.registry.* ;
4.  public class UserDao {
5.   public static int register(User u){
6.   int i=0;
7.  StandardServiceRegistry ssr = new StandardServiceRegistryBuilder().configure("hibe
    rnate.cfg.xml").build();
8.   Metadata meta = new MetadataSources(ssr).getMetadataBuilder().build();
9.   SessionFactory factory = meta.getSessionFactoryBuilder().build();
10. Session session = factory.openSession();
11. Transaction t = session.beginTransaction();
12.  i=(Integer)session.save(u);
13.  t.commit();
14. session.close();
15.  return i;
16.  }
17. }
```

hibernate.cfg.xml

```
1. <?xml version='1.0' encoding='UTF-8'?>
2. <!DOCTYPE hibernate-configuration PUBLIC
3.         "-//Hibernate/Hibernate Configuration DTD 5.3//EN"
4.         "http://hibernate.sourceforge.net/hibernate-configuration-5.3.dtd">
5. <hibernate-configuration>
6.    <session-factory>
7.       <property name="hbm2ddl.auto">update</property>
8.       <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
9.     <property name="connection.url">jdbc:mysql://localhost:3306/Test</property>
10.    <property name="connection.username">root</property>
11.     <property name="connection.password">root</property>
12.    <property name="connection.driver_class"> com.mysql.jdbc.Driver</property>
13.   <mapping resource="user.hbm.xml"/>
14.   </session-factory>
15. </hibernate-configuration>
```

Hibernate Generator classes

- The **<generator> class** is a sub-element of id.
- It is used to generate the unique identifier for the objects of persistent class.
- There are many generator classes defined in the Hibernate Framework.
- All the generator classes implements the **org.hibernate.id.IdentifierGenerator interface.**
- Hibernate framework provides many built-in generator classes:

For example:

```
<id ...>
  <generator class="type"></generator>
</id>
```

1. **assigned:** the default generator strategy if there is no <generator> element .
2. **increment:** uses for the auto increment field of DB. It generates **short, int or long type** identifier.
3. **sequence:** uses sequence of the database. (i.e s1, s2...)
4. **hilo:** It uses high and low algorithm to generate the id of type **short, int and long**.
5. **identity:** It is used in Sybase, MySQL, MS SQL Server, DB2 etc. to support the id column. (short, int or long).
6. **native**: It uses identity, sequence or hilo depending on the database vendor.
7. **seqhilo:** combination of Sequence and hilo
8. **uuid:** generate 128 bit Universal Unique ID
9. **guid:** genertated by DB of type String specially used with MySql

10. **select:** It uses the primary key returned by the database trigger.
11. **foreign:** It uses the id of another associated object.
12. **sequence-identity:** It uses a special sequence generation strategy. It is supported in Oracle 10g drivers only.

Hibernate Dialects

- For connecting any hibernate application with the database, you must specify the SQL dialects.
- Dialects classes defined for RDBMS in the org.hibernate.dialect package.
- Few of them are as follows:

RDBMS	Dialect
Oracle (any version)	org.hibernate.dialect.OracleDialect
Oracle9i	org.hibernate.dialect.Oracle9iDialect
Oracle10g	org.hibernate.dialect.Oracle10gDialect
MySQL	org.hibernate.dialect.MySQLDialect
DB2	org.hibernate.dialect.DB2Dialect
Microsoft SQL Server	org.hibernate.dialect.SQLServerDialect

Hibernate with Log4j1

- Logging enables the programmer to write the log details into a file permanently.
- Log4j and Logback frameworks can be used in hibernate framework to support logging.
- There are two ways to perform logging using log4j:
    - By log4j.xml file (or)
    - By log4j.properties file

Levels of Logging

- Following are the common logging levels.

Levels	Description
OFF	This level is used to turn off logging.
WARNING	This is a message level that indicates a problem.
SEVERE	This is a message level that indicates a failure.
INFO	This level is used for informational messages.
CONFIG	This level is used for static configuration messages.

Steps to perform Hibernate Logging by Log4j using xml file

- There are two ways to perform logging using log4j using xml file:
    1. Load the log4j jar files with hibernate
    2. Create the log4j.xml file inside the src folder (parallel with hibernate.cfg.xml file)

Example of Hibernate Logging by Log4j using xml file

- You can enable logging in hibernate by following only two steps in any hibernate example.
- This is the first example of hibernate application with logging support using log4j.

Load the required jar files

- You need to load the slf4j.jar and log4j.jar files with hibernate jar files.

Create log4j.xml file

- Now you need to create log4j.xml file.

log4j.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/"
    debug="false">
<appender name="CONSOLE" class="org.apache.log4j.ConsoleAppender">
 <layout class="org.apache.log4j.PatternLayout">
  <param name="ConversionPattern" value="[%d{dd/MM/yy hh:mm:ss:sss z}] %5p
%c{2}: %m%n" />  </layout>  </appender>
    <appender name="ASYNC" class="org.apache.log4j.AsyncAppender">
        <appender-ref ref="CONSOLE" />
        <appender-ref ref="FILE" />  </appender>
<appender name="FILE" class="org.apache.log4j.RollingFileAppender">
    <param name="File" value="C:/test.log" />
    <param name="MaxBackupIndex" value="100" />
 <layout class="org.apache.log4j.PatternLayout">
  <param name="ConversionPattern" value="[%d{dd/MM/yy hh:mm:ss:sss z}] %5p
%c{2}: %m%n" />  </layout>  </appender>
    <category name="org.hibernate">
        <priority value="DEBUG" />
    </category>
    <category name="java.sql">
        <priority value="debug" />
    </category>
    <root>
        <priority value="INFO" />
        <appender-ref ref="FILE" />      </root>  </log4j:configuration>
```

Hibernate with Log4j2

- As we know, Log4j and Logback frameworks are used to support logging in hibernate, there are two ways to perform logging using log4j:

- By log4j.xml file (or)
- By log4j.properties file

- Here, we are going to enable logging using log4j through properties file.

Steps to perform Hibernate Logging by Log4j using properties file

- There are two ways to perform logging using log4j using properties file:
  1. Load the log4j jar files with hibernate
  2. Create the log4j.properties file inside the src folder (parallel with hibernate.cfg.xml file)

Example using properties file

- You can enable logging in hibernate by following only two steps in any hibernate example.
- This is the first example of hibernate application with logging support using log4j.

Load the required jar files

- You need to load the slf4j.jar and log4j.jar files with hibernate jar files.

Create log4j.properties file

- Now you need to create log4j.properties file. In this example, all the log details will be written in the C:\\test.log file.

log4j.properties

```
1.  # Direct log messages to a log file
2.  log4j.appender.file=org.apache.log4j.RollingFileAppender
3.  log4j.appender.file.File=C:\\test.log
4.  log4j.appender.file.MaxFileSize=1MB
5.  log4j.appender.file.MaxBackupIndex=1
6.  log4j.appender.file.layout=org.apache.log4j.PatternLayout
7.  log4j.appender.file.layout.ConversionPattern=%d{ABSOLUTE} %5p %c{1}:%L -
     %m%n
8.   # Direct log messages to stdout
9.       log4j.appender.stdout=org.apache.log4j.ConsoleAppender
10.      log4j.appender.stdout.Target=System.out
11.      log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
12.      log4j.appender.stdout.layout.ConversionPattern=%d{ABSOLUTE} %5p %c{1}:
         %L - %m%n
13.      # Root logger option
14.      log4j.rootLogger=INFO, file, stdout
15.      # Log everything. Good for troubleshooting
16.      log4j.logger.org.hibernate=INFO
17.      # Log all JDBC parameters
18.      log4j.logger.org.hibernate.type=ALL
```

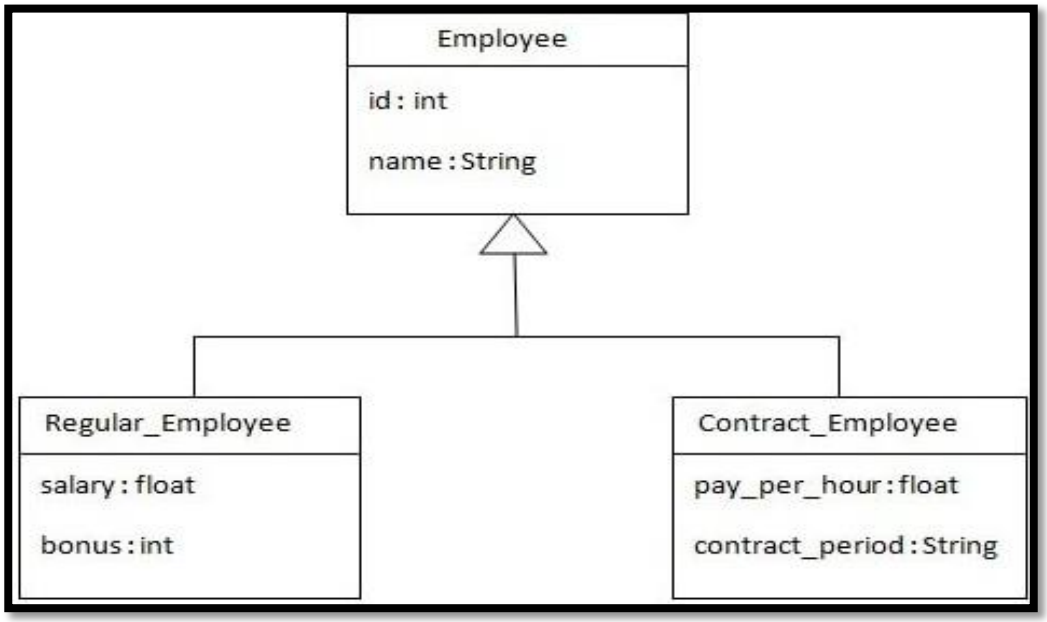
Unit 4 - 5

Inheritance Mapping

- We can map the inheritance hierarchy classes with the table of the database. There are three inheritance mapping strategies defined in the hibernate:
  1. **Table Per Hierarchy**
     In table per hierarchy mapping, single table is required to map the whole hierarchy, an extra column is added to identify the class. But nullable values are stored in the table .
     - Table Per Hierarchy using xml file
     - Table Per Hierarchy using Annotation
  2. **Table Per Concrete class**
     In case of table per concrete class, tables are created as per class. But duplicate column is added in subclass tables.
     - Table Per Concrete class using xml file
     - Table Per Concrete class using Annotation
  3. **Table Per Subclass**
     In this strategy, tables are created as per class but related by foreign key. So there are no duplicate columns.
     - Table Per Subclass using xml file
     - Table Per Subclass using Annotation

Hibernate Table Per Hierarchy using xml file

- By this inheritance strategy, we can map the whole hierarchy by single table only.
- Here, an extra column known as **discriminator column** is created in the table to identify the class.
- Let's understand the problem first.



- There are three classes in this hierarchy.
- Employee is the super class for Regular_Employee and Contract_Employee classes.
- In case of table per class hierarchy an discriminator column is added by the hibernate framework that specifies the type of the record.
- It is mainly used to distinguish the record.
- To specify this, discriminator subelement of class must be specified.
- The subclass subelement of class, specifies the subclass.
- In this case, Regular_Employee and Contract_Employee are the subclasses of Employee class.
- The table structure for this hierarchy is as shown below:

Column Name	Data Type	Nullable	Default	Primary Key
ID	NUMBER(10,0)	No	-	1
TYPE	VARCHAR2(255)	No	-	-
NAME	VARCHAR2(255)	Yes	-	-
SALARY	FLOAT	Yes	-	-
BONUS	NUMBER(10,0)	Yes	-	-
PAY_PER_HOUR	FLOAT	Yes	-	-
CONTRACT_DURATION	VARCHAR2(255)	Yes	-	-
				1 - 7

Employee.java

```
1. public class Employee {
2. private int id;
3. private String name;
4.   //getters and setters
5. }
```

Regular_Employee.java

```
1. public class Regular_Employee extends Employee{
2. private float salary;
3. private int bonus;
4.   //getters and setters
5. }
```

Contract_Employee.java

```
1. public class Contract_Employee extends Employee{
2.     private float pay_per_hour;
3.     private String contract_duration;
4.   //getters and setters
5. }
```

employee.hbm.xml

```
1.  <?xml version='1.0' encoding='UTF-8'?>
2.  <!DOCTYPE hibernate-mapping PUBLIC
3.          "-//Hibernate/Hibernate Mapping DTD 5.3//EN"
4.          "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
5.   <hibernate-mapping>
6.  <class name=" Employee" table="emp" discriminator-value="emp">
7.  <id name="id"> <generator class="increment"></generator>  </id>
8.   <discriminator column="type" type="string"></discriminator>
9.  <property name="name"></property>
10.  <subclass name=" Regular_Employee" discriminator-value="reg_emp">
11.    <property name="salary"></property>
12.    <property name="bonus"></property>
13. </subclass>
14. <subclass name="Contract_Employee" discriminator-value="con_emp">
15.    <property name="pay_per_hour"></property>
16.    <property name="contract_duration"></property>
17. </subclass>
18.  </class>  </hibernate-mapping>
```

hibernate.cfg.xml

```
1.  <?xml version='1.0' encoding='UTF-8'?>
2.  <!DOCTYPE hibernate-configuration PUBLIC
3.          "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
4.          "http://hibernate.sourceforge.net/hibernate-configuration-3.0 dtd">
5.  <hibernate-configuration>
6.    <session-factory>
7.       <property name="hbm2ddl.auto">update</property>
8.       <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
9.    <property name="connection.url">jdbc:mysql://localhost:3306/Test</property>
10.    <property name="connection.username">root</property>
11.     <property name="connection.password">root</property>
12.    <property name="connection.driver_class">
    com.mysql.jdbc.Driver</property>
13.    <mapping resource="employee.hbm.xml"/>
14.    </session-factory>
15. </hibernate-configuration>
```

StoreData.java

```
1.  import org.hibernate.*;
2.  import org.hibernate.cfg.*;
3.   public class StoreData {
4.  public static void main(String[] args) {
5.       Configuration cfg=new Configuration();
6.       cfg.configure("hibernate.cfg.xml");
7.       SessionFactory factory=cfg.buildSessionFactory();
8.        Session session=factory.openSession();
9.                Transaction tx=session.beginTransaction();
10.
11.    Employee e1=new Employee();
12.    e1.setName("Gaurav Chawla");
13.    Regular_Employee e2=new Regular_Employee();
14.    e2.setName("Vivek Kumar");
15.    e2.setSalary(50000);
16.    e2.setBonus(5);
17.   Contract_Employee e3=new Contract_Employee();
18.    e3.setName("Arjun Kumar");
19.    e3.setPay_per_hour(1000);
20.    e3.setContract_duration("15 hours");
21.    session.persist(e1);
22.    session.persist(e2);
23.    session.persist(e3);
24.    t.commit();
25.    session.close();
26.    System.out.println("success");
27. } }
```

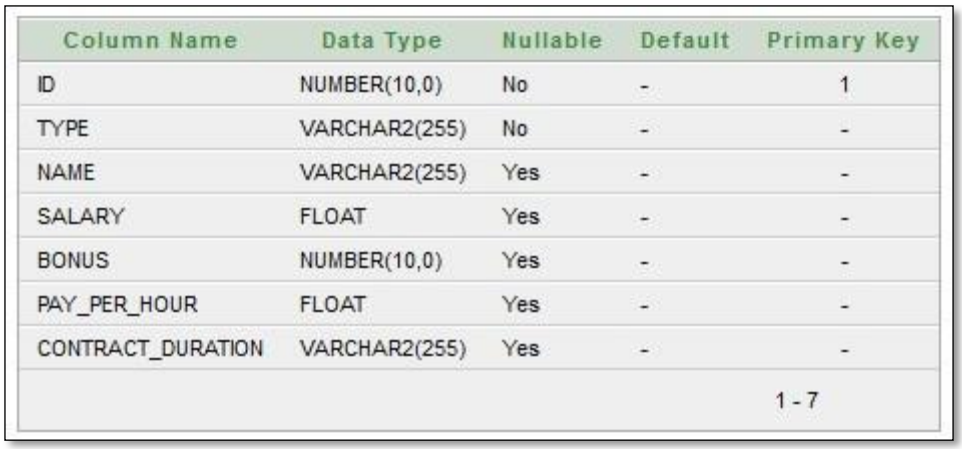
Output:

ID	TYPE	NAME	SALARY	BONUS	PAY_PER_HOUR	CONTRACT_DURATION
1	emp	Gaurav Chawla	-	-	-	-
2	reg_emp	Vivek Kumar	50000	5	-	-
3	con_emp	Arjun Kumar	-	-	1000	15 hours

Hibernate Table Per Hierarchy using Annotation

- Here, we are going to perform above example using annotation.
- You need to use @Inheritance(strategy=InheritanceType.SINGLE_TABLE), @DiscriminatorColumn and @DiscriminatorValue annotations for mapping table per hierarchy strategy.
- In case of table per hierarchy, only one table is required to map the inheritance hierarchy.

- Here, an extra column (also known as **discriminator column**) is created in the table to identify the class.



Column Name	Data Type	Nullable	Default	Primary Key
ID	NUMBER(10,0)	No	-	1
TYPE	VARCHAR2(255)	No	-	-
NAME	VARCHAR2(255)	Yes	-	-
SALARY	FLOAT	Yes	-	-
BONUS	NUMBER(10,0)	Yes	-	-
PAY_PER_HOUR	FLOAT	Yes	-	-
CONTRACT_DURATION	VARCHAR2(255)	Yes	-	-

1 - 7

Employee.java

```
1. import javax.persistence.*;
2. @Entity
3. @Table(name = "employee101")
4. @Inheritance(strategy=InheritanceType.SINGLE_TABLE)
5. @DiscriminatorColumn
       (name="type",discriminatorType=DiscriminatorType.STRING)
6. @DiscriminatorValue(value="employee")
7. public class Employee {
8. @Id
9. @GeneratedValue(strategy=GenerationType.AUTO)
10. @Column(name = "id")
11. private int id;
12. @Column(name = "name")
13. private String name;
14. //setters and getters
15. }
```

Regular_Employee.java

```
1. import javax.persistence.*;
2. @Entity
3. @DiscriminatorValue("regularemployee")
4. public class Regular_Employee extends Employee{
5. @Column(name="salary")
6. private float salary;
7. @Column(name="bonus")
8. private int bonus;
9. //setters and getters
10. }
```

Contract_Employee.java

```
1. import javax.persistence.Column;
2. import javax.persistence.DiscriminatorValue;
3. import javax.persistence.Entity;
4. @Entity
5. @DiscriminatorValue("contractemployee")
6. public class Contract_Employee extends Employee{
7.     @Column(name="pay_per_hour")
8.     private float pay_per_hour;
9.     @Column(name="contract_duration")
10.    private String contract_duration;
11.     //setters and getters
12. }
```

hibernate.cfg.xml

```
1. <?xml version='1.0' encoding='UTF-8'?>
2. <!DOCTYPE hibernate-configuration PUBLIC
3.           "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
4.           "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
5. <hibernate-configuration>
6.    <session-factory>
7.       <property name="hbm2ddl.auto">update</property>
8.       <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
9.      <property
   name="connection.url">jdbc:mysql://localhost:3306/Employee</property>
10.     <property name="connection.username">root</property>
11.      <property name="connection.password"></property>
12.     <property name="connection.driver_class">
   com.mysql.jdbc.Driver</property>
13.   <mapping class="Employee"/>
14.    <mapping class="Regular_Employee"/>
15.       <mapping class="Contract_Employee"/>
16.    </session-factory>
17. </hibernate-configuration>
```

StoreData.java

```
1. import org.hibernate.Session;
2. import org.hibernate.Transaction;
3. import org.hibernate.cfg.AnnotationConfiguration;
4. public class StoreData
5. {
6.     public static void main(String[] args)
7.       {
```

```
8.      Session session = new
    AnnotationConfiguration().configure().buildSessionFactory().openSession();
9.      Transaction tx=session.beginTransaction();
10.
11.      Employee e1=new Employee();
12.      e1.setName("Dhaval Sagathiya");
13.
14.      Regular_Employee e2=new Regular_Employee();
15.      e2.setName("Nitin Makawana");
16.      e2.setSalary(50000);
17.      e2.setBonus(5);
18.
19.     Contract_Employee e3=new Contract_Employee();
20.      e3.setName("Kavya Parmar");
21.      e3.setPay_per_hour(1000);
22.      e3.setContract_period("15 hours");
23.
24.      session.persist(e1);
25.      session.persist(e2);
26.      session.persist(e3);
27.
28.      tx.commit();
29.      session.close();
30.      System.out.println("success");
31.     }
32. }
```

Hibernate Table Per Concrete using xml file

- In case of Table Per Concrete class, there will be three tables in the database having no relations to each other.
- There are two ways to map the table with table per concrete class strategy.
    - By union-subclass element
    - By self creating the table for each class
- In case of table per concrete class, there will be three tables in the database, each representing a particular class.
- The union-subclass subelement of class, specifies the subclass. It adds the columns of parent table into this table. In other words, it is working as a union.
- The table structure for each table will be as follows:

Employee class

Column Name	Data Type	Nullable	Default	Primary Key
ID	NUMBER(10,0)	No	-	1
NAME	VARCHAR2(255)	Yes	-	-
				1 - 2

Regular_Employee class

Column Name	Data Type	Nullable	Default	Primary Key
ID	NUMBER(10,0)	No	-	1
NAME	VARCHAR2(255)	Yes	-	-
SALARY	FLOAT	Yes	-	-
BONUS	NUMBER(10,0)	Yes	-	-
				1 - 4

Contract_Employee class

Column Name	Data Type	Nullable	Default	Primary Key
ID	NUMBER(10,0)	No	-	1
NAME	VARCHAR2(255)	Yes	-	-
PAY_PER_HOUR	FLOAT	Yes	-	-
CONTRACT_DURATION	VARCHAR2(255)	Yes	-	-
				1 - 4

Example of Table per concrete class

File: Employee.java

```
1. public class Employee {
2. private int id;
3. private String name;
4.   //getters and setters
5. }
```

File: Regular_Employee.java

```
1. public class Regular_Employee extends Employee{
2. private float salary;
3. private int bonus;
4.   //getters and setters
5. }
```

File: Contract_Employee.java

```
1. public class Contract_Employee extends Employee{
2.     private float pay_per_hour;
3.     private String contract_duration;
4.  //getters and setters  }
```

File: employee.hbm.xml

```
1. <?xml version='1.0' encoding='UTF-8'?>
2. <!DOCTYPE hibernate-mapping PUBLIC
3.          "-//Hibernate/Hibernate Mapping DTD 5.3//EN"
4.          "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
5.      <hibernate-mapping>
6.   <class name="Employee" table="emp122">
7.   <id name="id">
8.   <generator class="increment"></generator>
9.   </id>
10.    <property name="name"></property>
11.  <union-subclass name=" Regular_Employee" table="regemp122">
12.  <property name="salary"></property>
13.  <property name="bonus"></property>
14.  </union-subclass>
15.  <union-subclass name="Contract_Employee" table="contemp122">
16.  <property name="pay_per_hour"></property>
17.  <property name="contract_duration"></property>
18.  </union-subclass>
19.  </class>
20.  </hibernate-mapping>
```

File: hibernate.cgf.xml file

```
1. <?xml version='1.0' encoding='UTF-8'?>
2. <!DOCTYPE hibernate-configuration PUBLIC
3.          "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
4.          "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
5. <hibernate-configuration>
6.    <session-factory>
7.       <property name="hbm2ddl.auto">update</property>
8.       <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
9.    <property name="connection.url">jdbc:mysql://localhost:3306/Test</property>
10.    <property name="connection.username">root</property>
11.  <property name="connection.password"></property>
12.  <property name="connection.driver_class"> com.mysql.jdbc.Driver
     </property>
13.    <mapping resource="employee.hbm.xml"/>
14.    </session-factory>  </hibernate-configuration>
```

File: StoreData.java

```
1. import org.hibernate.*;
2. import org.hibernate.cfg.*;
3.  public class StoreData {
4.    public static void main(String[] args) {
5.     Configuration cfg=new Configuration();
6.     cfg.configure("hibernate.cfg.xml");
```

```
7.       SessionFactory factory=cfg.buildSessionFactory();
8.        Session session=factory.openSession();
9.                Transaction tx=session.beginTransaction();
1.      Employee e1=new Employee();
2.      e1.setName("Gaurav Chawla");
3.
4.      Regular_Employee e2=new Regular_Employee();
5.      e2.setName("Vivek Kumar");
6.      e2.setSalary(50000);
7.      e2.setBonus(5);
8.
9.      Contract_Employee e3=new Contract_Employee();
10.     e3.setName("Arjun Kumar");
11.     e3.setPay_per_hour(1000);
12.     e3.setContract_duration("15 hours");
13.
14.     session.persist(e1);
15.     session.persist(e2);
16.     session.persist(e3);
17.
18.     t.commit();
19.     session.close();
20.     System.out.println("success");
21. }
22. }
```

Example of TPC using Annotation

File: Employee.java

```
1.  import javax.persistence.*;
2.  @Entity
3.  @Table(name = "employee102")
4.  @Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
5.  public class Employee
6.  {
7.  @Id
8.  @GeneratedValue(strategy=GenerationType.AUTO)
9.  @Column(name = "id")
10. private int id;
11.
12. @Column(name = "name")
13. private String name;
14.
15. //setters and getters
16. }
```

File: Regular_Employee.java

```
1. import javax.persistence.*;
2. @Entity
3. @Table(name="regularemployee102")
4. @AttributeOverrides({
5.     @AttributeOverride(name="id", column=@Column(name="id")),
6.     @AttributeOverride(name="name", column=@Column(name="name"))
7. })
8. public class Regular_Employee extends Employee
9. {
10. @Column(name="salary")
11. private float salary;
12.
13. @Column(name="bonus")
14. private int bonus;
15.
16. //setters and getters
17. }
```

File: Contract_Employee.java

```
1. import javax.persistence.*;
2. @Entity
3. @Table(name="contractemployee102")
4. @AttributeOverrides({
5.     @AttributeOverride(name="id", column=@Column(name="id")),
6.     @AttributeOverride(name="name", column=@Column(name="name"))
7. })
8. public class Contract_Employee extends Employee{
9.       @Column(name="pay_per_hour")
10.    private float pay_per_hour;
11.
12.    @Column(name="contract_duration")
13.    private String contract_duration;
14.
15.    public float getPay_per_hour() {
16.       return pay_per_hour;
17.    }
18.    public void setPay_per_hour(float payPerHour) {
19.       pay_per_hour = payPerHour;
20.    }
21.    public String getContract_duration() {
22.       return contract_duration;
23.    }
24.    public void setContract_duration(String contractDuration) {
25.       contract_duration = contractDuration;
26.    }
27. }
```

hibernate.cfg.xml

```
1.  <?xml version='1.0' encoding='UTF-8'?>
2.  <!DOCTYPE hibernate-configuration PUBLIC
3.            "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
4.            "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
5.  <hibernate-configuration>
6.     <session-factory>
7.          <property name="hbm2ddl.auto">update</property>
8.          <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
9.        <property
    name="connection.url">jdbc:mysql://localhost:3306/Employee</property>
10.      <property name="connection.username">root</property>
11.       <property name="connection.password"></property>
12.      <property name="connection.driver_class">
    com.mysql.jdbc.Driver</property>
13.    <mapping class="Employee"/>
14.     <mapping class="Regular_Employee"/>
15.          <mapping class="Contract_Employee"/>
16.     </session-factory>
17. </hibernate-configuration>
```

StoreData.java

```
1.  import org.hibernate.Session;
2.  import org.hibernate.Transaction;
3.  import org.hibernate.cfg.AnnotationConfiguration;
4.  public class StoreData
5.  {
6.      public static void main(String[] args)
7.        {
8.        Session session = new
    AnnotationConfiguration().configure().buildSessionFactory().openSession();
9.        Transaction tx=session.beginTransaction();
10.         Employee e1=new Employee();
11.         e1.setName("Dhaval Shah");
12.         Regular_Employee e2=new Regular_Employee();
13.         e2.setName("Hardik Makawana");
14.         e2.setSalary(50000);
15.         e2.setBonus(5);
16.        Contract_Employee e3=new Contract_Employee();
17.         e3.setName("Dhruv Parekh");
18.         e3.setPay_per_hour(1000);
19.         e3.setContract_period("15 hours");
20.
21.         session.persist(e1);
```

```
22.         session.persist(e2);
23.         session.persist(e3);
24.
25.         tx.commit();
26.         session.close();
27.         System.out.println("success");
28.       }
29. }
```

Table Per Subclass Example using xml file

- In case of Table Per Subclass, subclass mapped tables are related to parent class mapped table by primary key and foreign key relationship.
- The **<joined-subclass>** element of class is used to map the child class with parent using the primary key and foreign key relation.
- In this example, we are going to use hb2ddl.auto property to generate the table automatically. So we don't need to be worried about creating tables in the database.

Table structure for Employee class

Column Name	Data Type	Nullable	Default	Primary Key
ID	NUMBER(10,0)	No	-	1
NAME	VARCHAR2(255)	Yes	-	-
				1 - 2

Table structure for Regular_Employee class

Column Name	Data Type	Nullable	Default	Primary Key
EID	NUMBER(10,0)	No	-	1
SALARY	FLOAT	Yes	-	-
BONUS	NUMBER(10,0)	Yes	-	-
				1 - 3

Table structure for Contract_Employee class

Column Name	Data Type	Nullable	Default	Primary Key
EID	NUMBER(10,0)	No	-	1
PAY_PER_HOUR	FLOAT	Yes	-	-
CONTRACT_DURATION	VARCHAR2(255)	Yes	-	-
				1 - 3

Example :

File: Employee.java

```
1. public class Employee {
2. private int id;
3. private String name;
4.
5. //getters and setters
6. }
```

File: Regular_Employee.java

```
1. public class Regular_Employee extends Employee{
2. private float salary;
3. private int bonus;
4.
5. //getters and setters
6. }
```

File: Contract_Employee.java

```
1. public class Contract_Employee extends Employee{
2.     private float pay_per_hour;
3.     private String contract_duration;
4.
5. //getters and setters
6. }
```

File: employee.hbm.xml

```
1. <?xml version='1.0' encoding='UTF-8'?>
2. <!DOCTYPE hibernate-mapping PUBLIC
3.    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
4.   "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
5.  <hibernate-mapping>
6.   <class name=" Employee" table="emp123">
7.   <id name="id">
8.   <generator class="increment"></generator>
9.   </id>
10.
11.  <property name="name"></property>
12.
13.  <joined-subclass name="Regular_Employee" table="regemp123">
14.  <key column="eid"></key>
15.  <property name="salary"></property>
16.  <property name="bonus"></property>
17.  </joined-subclass>
18.
19.  <joined-subclass name="Contract_Employee" table="contemp123">
20.  <key column="eid"></key>
21.  <property name="pay_per_hour"></property>
22.  <property name="contract_duration"></property>
23.  </joined-subclass>
24.   </class>   </hibernate-mapping>
```

File: hibernate.cgf.xml

```
1.  <?xml version='1.0' encoding='UTF-8'?>
2.  <!DOCTYPE hibernate-configuration PUBLIC
3.          "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
4.          "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
5.  <hibernate-configuration>
6.    <session-factory>
7.       <property name="hbm2ddl.auto">update</property>
8.       <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
9.     <property name="connection.url">jdbc:mysql://localhost:3306/Test</property>
10.    <property name="connection.username">root</property>
11.   <property name="connection.password"></property>
12. <property name="connection.driver_class"> com.mysql.jdbc.Driver </property>
13.    <mapping resource="employee.hbm.xml"/>
14.    </session-factory>  </hibernate-configuration>
```

File: StoreData.java

```
1.  import org.hibernate.*;
2.  import org.hibernate.cfg.*;
3.    public class StoreData {
4.      public static void main(String[] args) {
5.       Configuration cfg=new Configuration();
6.       cfg.configure("hibernate.cfg.xml");
7.       SessionFactory factory=cfg.buildSessionFactory();
8.        Session session=factory.openSession();
9.        Transaction tx=session.beginTransaction();
10.      Employee e1=new Employee();
11.      e1.setName("Gaurav Chawla");
12.       Regular_Employee e2=new Regular_Employee();
13.       e2.setName("Vivek Kumar");
14.       e2.setSalary(50000);
15.       e2.setBonus(5);
16.     Contract_Employee e3=new Contract_Employee();
17.       e3.setName("Arjun Kumar");
18.       e3.setPay_per_hour(1000);
19.       e3.setContract_duration("15 hours");
20.       session.persist(e1);
21.       session.persist(e2);
22.       session.persist(e3);
23.       tx.commit();
24.       session.close();
25.       System.out.println("success");
26.    }  }
```

Table Per Subclass using Annotation

Example

File: Employee.java

```
1. import javax.persistence.*;
2. @Entity
3. @Table(name = "employee103")
4. @Inheritance(strategy=InheritanceType.JOINED)
5.
6. public class Employee {
7. @Id
8. @GeneratedValue(strategy=GenerationType.AUTO)
9.
10. @Column(name = "id")
11. private int id;
12.
13. @Column(name = "name")
14. private String name;
15.
16. //setters and getters
17. }
```

File: Regular_Employee.java

```
1. import javax.persistence.*;
2.
3. @Entity
4. @Table(name="regularemployee103")
5. @PrimaryKeyJoinColumn(name="ID")
6. public class Regular_Employee extends Employee{
7.
8. @Column(name="salary")
9. private float salary;
10.
11. @Column(name="bonus")
12. private int bonus;
13.
14. //setters and getters
15. }
```

File: Contract_Employee.java

```
1. import javax.persistence.*;
2.
3. @Entity
4. @Table(name="contractemployee103")
5. @PrimaryKeyJoinColumn(name="ID")
6. public class Contract_Employee extends Employee{
7.
8.     @Column(name="pay_per_hour")
9.     private float pay_per_hour;
10.
```

```
11.    @Column(name="contract_duration")
12.    private String contract_duration;
13.
14.    //setters and getters
15. }
```

hibernate.cfg.xml

```
1.  <?xml version='1.0' encoding='UTF-8'?>
2.  <!DOCTYPE hibernate-configuration PUBLIC
3.            "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
4.            "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
5.  <hibernate-configuration>
6.     <session-factory>
7.          <property name="hbm2ddl.auto">update</property>
8.          <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
9.    <property name="connection.url">jdbc:mysql://localhost:3306/Test</property>
10.      <property name="connection.username">root</property>
11.       <property name="connection.password"></property>
12.   <property name="connection.driver_class"> com.mysql.jdbc.Driver</property>
13.    <mapping class="Employee"/>
14.     <mapping class="Regular_Employee"/>
15.         <mapping class="Contract_Employee"/>
16.     </session-factory>
17. </hibernate-configuration>
```

StoreData.java

```
1.  import org.hibernate.Session;
2.  import org.hibernate.Transaction;
3.  import org.hibernate.cfg.AnnotationConfiguration;
4.  public class StoreData
5.  {
6.      public static void main(String[] args)
7.        {
8.        Session session = new
    AnnotationConfiguration().configure().buildSessionFactory().openSession();
9.        Transaction tx=session.beginTransaction();
10.
11.         Employee e1=new Employee();
12.         e1.setName("Dhaval Sagathiya");
13.
14.         Regular_Employee e2=new Regular_Employee();
15.         e2.setName("Nitin Makawana");
16.         e2.setSalary(50000);
17.         e2.setBonus(5);
```

```
18.
19.       Contract_Employee e3=new Contract_Employee();
20.        e3.setName("Kavya Parmar");
21.        e3.setPay_per_hour(1000);
22.        e3.setContract_period("15 hours");
23.
24.        session.persist(e1);
25.        session.persist(e2);
26.        session.persist(e3);
27.
28.        tx.commit();
29.        session.close();
30.        System.out.println("success");
31.      }}
```

Collection Mapping in Hibernate

- We can map collection elements of Persistent class in Hibernate. You need to declare the type of collection in Persistent class from one of the following types:
  - java.util.List
  - java.util.Set
  - java.util.SortedSet
  - java.util.Map
  - java.util.SortedMap
  - java.util.Collection
- There are three subelements used in the list:
  - ✓ **<key>** element is used to define the foreign key in this table based
  - ✓ **<index>** element is used to identify the type. List and Map are indexed collection.
  - ✓ **<element>** is used to define the element of the collection.
- This is the mapping of collection if collection stores string objects.
- If collection stores entity reference (another class objects), we need to define <one-to-many> or <many-to-many> element.

Understanding key element

- The key element is used to define the foreign key in the joined table based on the original identity. The foreign key element is nullable by default. So for non-nullable foreign key, we need to specify not-null attribute such as:

```
<key column="qid" not-null="true" ></key>
```

- The attributes of the key element are column, on-delete, property-ref, not-null, update and unique.

Indexed collections

- The collection elements can be categorized in two forms: **indexed** ,and **non-indexed**

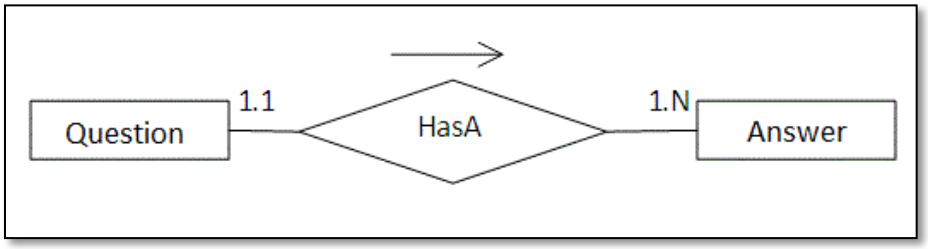
- The List and Map collection are indexed whereas set and bag collections are non-indexed. Here, indexed collection means List and Map requires an additional element **<index>**.

Collection Elements

- The collection elements can have value or entity reference (another class object).
- We can use one of the 4 elements
  - **element**
  - **component-element**
  - **one-to-many**
  - **many-to-many**
- The element and component-element are used for normal value such as string, int etc. whereas one-to-many and many-to-many are used to map entity reference.

Mapping List in Collection Mapping (using xml file)

- If our persistent class has List object, we can map the List easily either by <list> element of class in mapping file or by annotation.
- Here, we are using the scenario of Forum where one question has multiple answers.



Question.java

```
1. import java.util.List;
2. public class Question {
3. private int id;
4. private String qname;
5. private List<String> answers;
6. //getters and setters
7. }
```

question.hbm.xml

```
1. <?xml version='1.0' encoding='UTF-8'?>
2. <!DOCTYPE hibernate-mapping PUBLIC
3.         "-//Hibernate/Hibernate Mapping DTD 5.3//EN"
4.         "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
5.  <hibernate-mapping>
6.  <class name="com.javatpoint.Question" table="q100">
7.   <id name="id">
8.     <generator class="increment"></generator>
9.   </id>
10.  <property name="qname"></property>
```

```
    <list name="answers" table="ans100">
      <key column="qid"></key>
      <index column="type"></index>
      <element column="answer" type="string"></element>
    </list>
  </class>
</hibernate-mapping>
```

hibernate.cfg.xml

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
          "-//Hibernate/Hibernate Configuration DTD 5.3//EN"
          "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
      <property name="hbm2ddl.auto">update</property>
      <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
 <property name="connection.url">jdbc:mysql://localhost:3306/Test</property>
    <property name="connection.username">root</property>
     <property name="connection.password"></property>
  <property name="connection.driver_class"> com.mysql.jdbc.Driver</property>
   <mapping resource="question.hbm.xml"/>
   </session-factory>
</hibernate-configuration>
```

StoreData.java

```
import java.util.ArrayList;
import org.hibernate.*;
import org.hibernate.boot.*;
import org.hibernate.boot.registry.*
public class StoreData {
 public static void main(String[] args)
{
Session session = new   Configuration().configure("question.hbm.xml").
buildSessionFactory().openSession();
      Transaction tx=session.beginTransaction();

   ArrayList<String> list1=new ArrayList<String>();
   list1.add("Java is a programming language");
   list1.add("Java is a platform");

   ArrayList<String> list2=new ArrayList<String>();
   list2.add("Servlet is an Interface");
   list2.add("Servlet is an API");

   Question question1=new Question();
   question1.setQname("What is Java?");
```

```
21.     question1.setAnswers(list1);
22.
23.     Question question2=new Question();
24.     question2.setQname("What is Servlet?");
25.     question2.setAnswers(list2);
26.
27.     session.persist(question1);
28.     session.persist(question2);
29.
30.     t.commit();
31.     session.close();
32.     System.out.println("success");
33.  }
34. }
```

Output

ID	QNAME
1	What is Java?
2	What is Servlet?

QID	TYPE	ANSWER
1	0	Java is a programming language
1	1	Java is a platform
2	0	Servlet is an Interface
2	1	Servlet is an API

Mapping Bag in Collection Mapping (using xml file)

- If our persistent class has List object, we can map the List by list or bag element in the mapping file. The bag is just like List but it doesn't require index element.
- Here, we are using the scenario of Forum where one question has multiple answers.

Question.java

```
1.  import java.util.List;
2.  public class Question {
3.  private int id;
4.  private String qname;
5.  private List<String> answers;
6.  //getters and setters
7.  }
```

question.hbm.xml

```
1.  <?xml version='1.0' encoding='UTF-8'?>
2.  <!DOCTYPE hibernate-mapping PUBLIC
3.          "-//Hibernate/Hibernate Mapping DTD 5.3//EN"
4.          "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
5.   <hibernate-mapping>
6.   <class name="Question" table="q100">
7.    <id name="id">
8.      <generator class="increment"></generator>
9.    </id>
```

```
10.    <property name="qname"></property>
11.    <bag name="answers" table="ans100">
12.      <key column="qid"></key>
13.      <element column="answer" type="string"></element>
14.    </bag>
15.  </class>
16. </hibernate-mapping>
```

hibernate.cfg.xml

```
1.  <?xml version='1.0' encoding='UTF-8'?>
2.  <!DOCTYPE hibernate-configuration PUBLIC
3.            "-//Hibernate/Hibernate Configuration DTD 5.3//EN"
4.            "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
5.  <hibernate-configuration>
6.    <session-factory>
7.          <property name="hbm2ddl.auto">update</property>
8.          <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
9.   <property name="connection.url">jdbc:mysql://localhost:3306/Test</property>
10.       <property name="connection.username">root</property>
11.       <property name="connection.password"></property>
12.   <property name="connection.driver_class"> com.mysql.jdbc.Driver</property>
13.    <mapping resource="question.hbm.xml"/>
14.     </session-factory>
15. </hibernate-configuration>
```

StoreData.java

```
1.  import java.util.*;
2.  import org.hibernate.*;
3.  import org.hibernate.boot.*;
4.  import org.hibernate.boot.registry.*
5.  public class StoreData {
6.   public static void main(String[] args)
7.  {     Session session = new   Configuration().configure("question.hbm.xml").
    buildSessionFactory().openSession();
8.        Transaction tx=session.beginTransaction();
9.
10.     ArrayList<String> list1=new ArrayList<String>();
11.     list1.add("Java is a programming language");
12.     list1.add("Java is a platform");
13.
14.    Question question1=new Question();
15.     question1.setQname("What is Java?");
16.     question1.setAnswers(list1);
17.     session.persist(question1);
18.     t.commit();
19.     session.close();
20.     System.out.println("success");     }    }
```

Mapping Set in Collection Mapping (using xml file)

Question.java

```
1. import java.util.Set;
2. public class Question {
3. private int id;
4. private String qname;
5. private Set<String> answers;
6. //getters and setters
7. }
```

question.hbm.xml

```
1. <?xml version='1.0' encoding='UTF-8'?>
2. <!DOCTYPE hibernate-mapping PUBLIC
3.         "-//Hibernate/Hibernate Mapping DTD 5.3//EN"
4.         "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
5.  <hibernate-mapping>
6.  <class name=" Question" table="q100">
7.   <id name="id">
8.     <generator class="increment"></generator>
9.   </id>
10.  <property name="qname"></property>
11.  <set name="answers" table="ans100">
12.    <key column="qid"></key>
13.    <element column="answer" type="string"></element>
14.  </set>  </class>  </hibernate-mapping>
```

hibernate.cfg.xml

```
1. <?xml version='1.0' encoding='UTF-8'?>
2. <!DOCTYPE hibernate-configuration PUBLIC
3.         "-//Hibernate/Hibernate Configuration DTD 5.3//EN"
4.         "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
5. <hibernate-configuration>
6.   <session-factory>
7.       <property name="hbm2ddl.auto">update</property>
8.       <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
9.  <property name="connection.url">jdbc:mysql://localhost:3306/Test</property>
10.    <property name="connection.username">root</property>
11.     <property name="connection.password"></property>
12.  <property name="connection.driver_class"> com.mysql.jdbc.Driver</property>
13.   <mapping resource="question.hbm.xml"/>
14.    </session-factory>
15. </hibernate-configuration>
```

StoreData.java

```
1. import java.util.*;
2. import org.hibernate.*;
3. import org.hibernate.boot.*;
```

```
4. import org.hibernate.boot.registry.*
5. public class StoreData {
6.  public static void main(String[] args)
7. {     Session session = new   Configuration().configure("question.hbm.xml").
   buildSessionFactory().openSession();
8.       Transaction tx=session.beginTransaction();
9.      HashSet<String> list1=new  HashSet <String>();
10.     list1.add("Java is a programming language");
11.     list1.add("Java is a platform");
12.
13.     Question question1=new Question();
14.     question1.setQname("What is Java?");
15.     question1.setAnswers(list1);
16.
17.     session.persist(question1);
18.     t.commit();
19.     session.close();
20.     System.out.println("success");
21.  }
22. }
```

Mapping Map in Collection Mapping (using xml file)

Question.java

```
1. import java.util.*;
2. public class Question {
3. private int id;
4. private String qname;
5. private Map<String,String> answers;
6. //getters and setters
7. }
```

question.hbm.xml

```
1. <?xml version='1.0' encoding='UTF-8'?>
2. <!DOCTYPE hibernate-mapping PUBLIC
3.          "-//Hibernate/Hibernate Mapping DTD 5.3//EN"
4.          "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
5.   <hibernate-mapping>
6.   <class name=" Question" table="q100">
7.     <id name="id">
8.       <generator class="increment"></generator>
9.     </id>
10.    <property name="qname"></property>
11.    <map name="answers" table="ans100">
12.      <key column="qid"></key>
13.      <index column="answer" type="string">
```

```
14.     <element column="username" type="string"></element>
15.   </map>   </class>  </hibernate-mapping>
```

hibernate.cfg.xml

```
1.  <?xml version='1.0' encoding='UTF-8'?>
2.  <!DOCTYPE hibernate-configuration PUBLIC
3.            "-//Hibernate/Hibernate Configuration DTD 5.3//EN"
4.            "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
5.  <hibernate-configuration>
6.    <session-factory>
7.          <property name="hbm2ddl.auto">update</property>
8.          <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
9.    <property name="connection.url">jdbc:mysql://localhost:3306/Test</property>
10.       <property name="connection.username">root</property>
11.       <property name="connection.password"></property>
12.   <property name="connection.driver_class"> com.mysql.jdbc.Driver</property>
13.    <mapping resource="question.hbm.xml"/>
14.     </session-factory>
15. </hibernate-configuration>
```

StoreData.java

```
1.  import java.util.*;
2.  import org.hibernate.*;
3.  import org.hibernate.boot.*;
4.  import org.hibernate.boot.registry.*
5.  public class StoreData {
6.   public static void main(String[] args)
7.  {     Session session = new   Configuration().configure("question.hbm.xml").
    buildSessionFactory().openSession();
8.         Transaction tx=session.beginTransaction();
9.       HashMap<String, String> list1=new  HashMap <String,String>();
10.      list1.add("Java is a programming language","RP");
11.      list1.add("Java is a platform","KK");
12.
13.      Question question1=new Question();
14.      question1.setQname("What is Java?");
15.      question1.setAnswers(list1);
16.
17.      session.persist(question1);
18.
19.      t.commit();
20.      session.close();
21.      System.out.println("success");
22.  }
23. }
```

Hibernate Many to Many Example using XML

- We can map many to many relation either using list, set, bag, map, etc. Here, we are going to use **list for many-to-many mapping**. In such case, three tables will be created.
- In this example, we will generate a many to many relation between questions and answers using list.

Question.java

```
1. import java.util.List;
2. public class Question {
3. private int id;
4. private String qname;
5. private List<String> answers;
6. //getters and setters
7. }
```

Answer.java

```
1. import java.util.*;
2. public class Answer {
3. private int id;
4. private String answername;
5. private String postedBy;
6. private List<Question> questions;
7. //getters and setters
8. }
```

question.hbm.xml

```
1. <?xml version='1.0' encoding='UTF-8'?>
2. <!DOCTYPE hibernate-mapping PUBLIC
3.         "-//Hibernate/Hibernate Mapping DTD 5.3//EN"
4.         "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
5.  <hibernate-mapping>
6.  <class name="Question" table="ques1911">
7.         <id name="id" type="int">
8.            <column name="q_id" />
9.            <generator class="increment" />
10.        </id>
11.        <property name="qname" />
12.
          <list name="answers" table="ques_ans1911" fetch="select" cascade="all">
13.           <key column="q_id" />
14.             <index column="type"></index>
15.           <many-to-many class="Answer" column="ans_id" />
16.       </list>
17.    </class>
18. </hibernate-mapping>
```

answer.hbm.xml

```
1.  <?xml version='1.0' encoding='UTF-8'?>
2.  <!DOCTYPE hibernate-mapping PUBLIC
3.            "-//Hibernate/Hibernate Mapping DTD 5.3//EN"
4.            "http://hibernate.sourceforge.net/hibernate-mapping-5.3.dtd">
5.
6.  <hibernate-mapping>
7.  <class name="com.javatpoint.Answer" table="ans1911">
8.          <id name="id" type="int">
9.              <column name="ans_id" />
10.             <generator class="increment" />
11.         </id>
12.         <property name="answername"  />
13.         <property name="postedBy" />
14.     </class>
15. </hibernate-mapping>
```

hibernate.cfg.xml

```
1.  <?xml version='1.0' encoding='UTF-8'?>
2.  <!DOCTYPE hibernate-configuration PUBLIC
3.            "-//Hibernate/Hibernate Configuration DTD 5.3//EN"
4.            "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
5.  <hibernate-configuration>
6.    <session-factory>
7.          <property name="hbm2ddl.auto">update</property>
8.          <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
9.   <property name="connection.url">jdbc:mysql://localhost:3306/Test</property>
10.       <property name="connection.username">root</property>
11.        <property name="connection.password"></property>
12.   <property name="connection.driver_class"> com.mysql.jdbc.Driver</property>
13.    <mapping resource="question.hbm.xml"/>
14.     </session-factory>
15. </hibernate-configuration>
```

StoreData.java

```
1.  import java.util.*;
2.  import org.hibernate.*;
3.  import org.hibernate.boot.*;
4.  import org.hibernate.boot.registry.*
5.  public class StoreData {
6.   public static void main(String[] args)
7.  {
8.  Session session = new   Configuration().configure("question.hbm.xml").
    buildSessionFactory().openSession();
9.        Transaction tx=session.beginTransaction();
10.
11.    Answer ans1=new Answer();
12.    ans1.setAnswername("Java is a programming language");
13.    ans1.setPostedBy("Ravi Malik");
14.
```

```
15.  Answer ans2=new Answer();
16.  ans2.setAnswername("Java is a platform");
17.  ans2.setPostedBy("Sudhir Kumar");
18.
19.  Question q1=new Question();
20.  q1.setQname("What is Java?");
21.
22.  ArrayList<Answer> l1=new ArrayList<Answer>();
23.  l1.add(ans1);
24.  l1.add(ans2);
25.  q1.setAnswers(l1);
26.
27.  Answer ans3=new Answer();
28.  ans3.setAnswername("Servlet is an Interface");
29.  ans3.setPostedBy("Jai Kumar");
30.
31.  Answer ans4=new Answer();
32.  ans4.setAnswername("Servlet is an API");
33.  ans4.setPostedBy("Arun");
34.
35.  Question q2=new Question();
36.  q2.setQname("What is Servlet?");
37.  ArrayList<Answer> l2=new ArrayList<Answer>();
38.  l2.add(ans3);
39.  l2.add(ans4);
40.  q2.setAnswers(l2);
41.
42.  session.persist(q1);
43.  session.persist(q2);
44.     t.commit();
45.     session.close();
46.     System.out.println("success");
47.   }
48.  }
```