MODULE-12
SPRING, AJAX AND DESIGN PATTERNS

Course Topics

- → Module 1
  - » Introduction to Java
- → Module 2
  - » Data Handling and Functions
- → Module 3
  - » Object Oriented Programming in Java
- → Module 4
  - » Packages and Multi-threading
- → Module 5
  - » Collections
- → Module 6
  - » XML
- → Module 7
  - » JDBC
- → Module 8
  - » Servlets
- → Module 9
  - » JSP
- → Module 10
  - » Hibernate
- → Module 11
  - » Spring
- → Module 12
  - » **Spring, Ajax and Design Patterns**
- → Module 13
  - » SOA
- → Module 14
  - » Web Services and Project

Objectives

At the end of this module, you will be able to

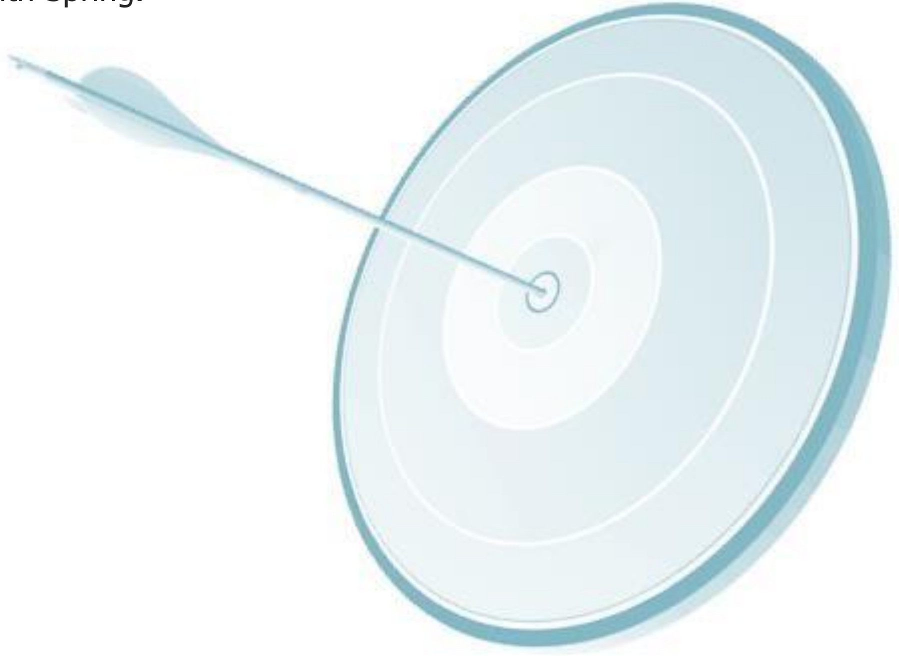
→ Understand what is AOP and how to integrate AOP with Spring.

→ Understand how to use JDBC with Spring

→ Learn how to integrate hibernate with Spring

→ Understand Ajax and its usage

→ Understand core and J2EE design patterns.



Spring AOP - Introduction

→ Aspect Oriented Programming provides modularity.

→ AOP breaks the logic into various parts which is called concerns. Using cross-cutting concerns modularity can be increased.

→ Intercepting filters can be used in AOP as filtering mechanism for user validation.

→ Advice is one of the cross-cutting concern of AOP. If a class implements Method BeforeAdvice and specified in the bean then before executing the actual bean, before method is invoked for filtering purpose. It is like filters in Servlets.

AOP Use Cases

Some of the use cases of AOP are:

→ Data encryption

→ Login validation

→ Logging

Spring AOP

→ Procedure to implement AOP Before Advice:

- » Write a bean class.
- » Write a class implementing MethodBeforeAdvice and implement a before(). This method will be invoked before the business logic bean is invoked.

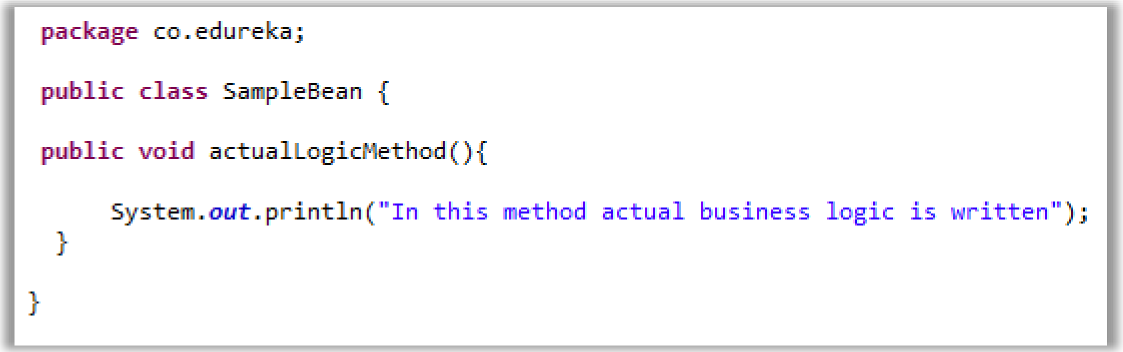
→ In the XML file:

- » Define a bean (Business logic).
- » Define a bean for the class which has implemented MethodBeforeAdvice interface.
- » Define ProxyFactoryBean and specify the business logic bean and intercepting filter bean.

→ In the client file:

- » Load the XML file.
- » Get the ProxyFactoryBean and invoke the methods of Business Logic Bean.
- » Here before() method would be called before executing the methods of Business Logic bean.

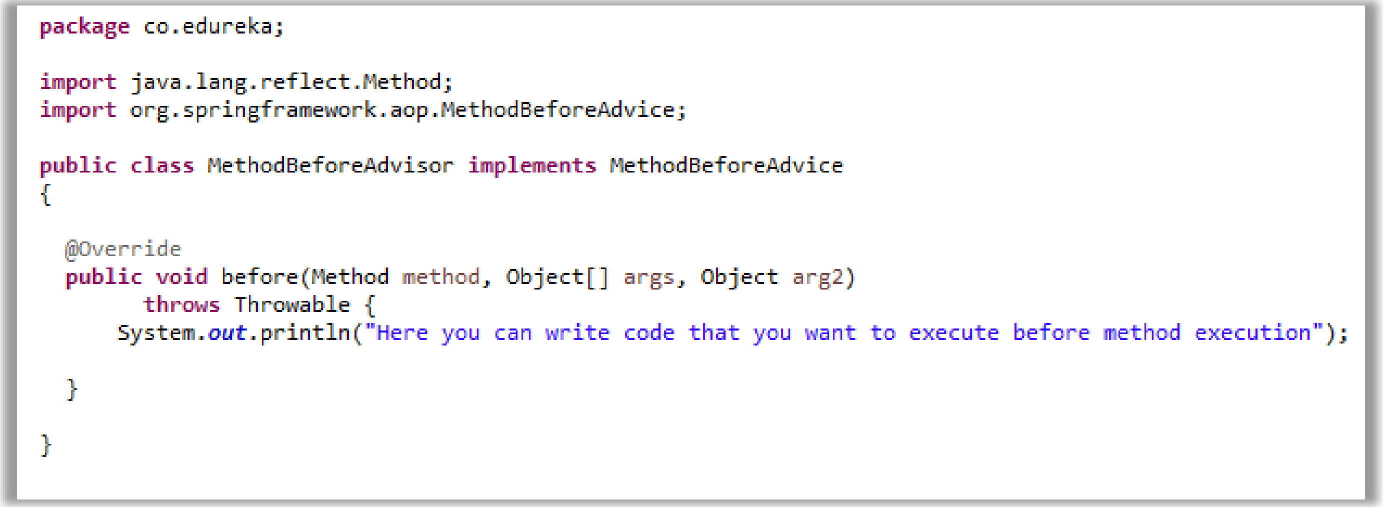
Spring AOP Example



```
package co.edureka;

public class SampleBean {

public void actualLogicMethod(){

        System.out.println("In this method actual business logic is written");
  }

}
```

This is SampleBean class where we have defined one method named actualLogicMethod.

Spring AOP Example (Contd.)

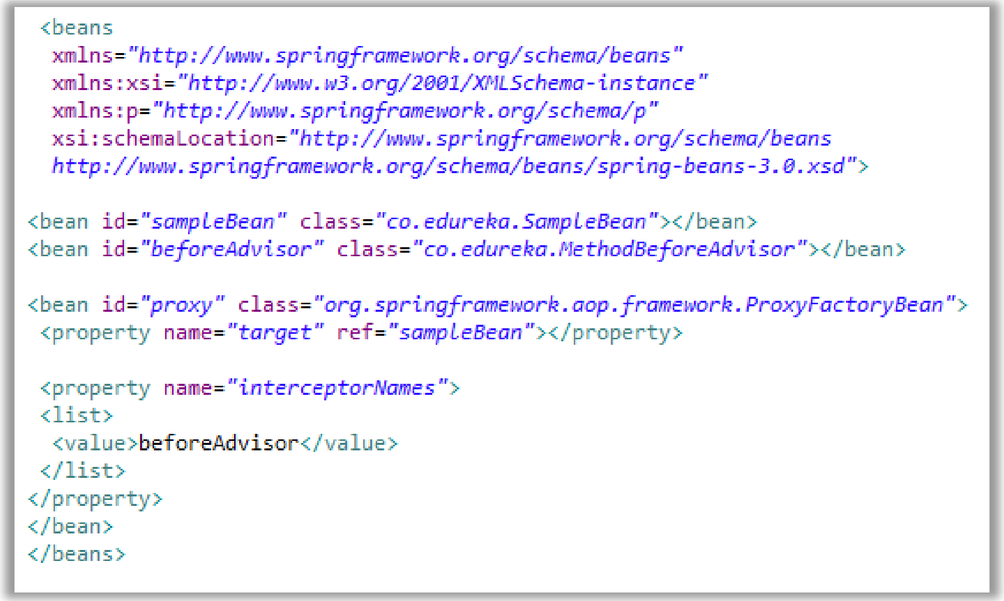


```
package co.edureka;

import java.lang.reflect.Method;
import org.springframework.aop.MethodBeforeAdvice;

public class MethodBeforeAdvisor implements MethodBeforeAdvice
{

  @Override
  public void before(Method method, Object[] args, Object arg2)
        throws Throwable {
      System.out.println("Here you can write code that you want to execute before method execution");

  }

}
```

Note that MethodBeforeAdvisor class implements MethodBeforeAdvice interface which have before method.

You can write any code that you want to execute before actual method is executed.

Spring AOP Example (Contd.)

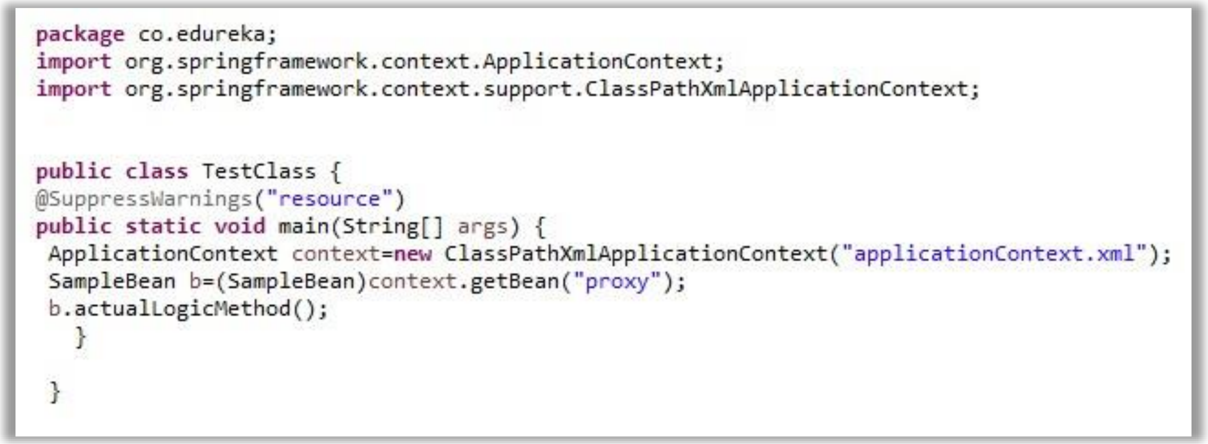


```
 <beans
  xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:p="http://www.springframework.org/schema/p"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<bean id="sampleBean" class="co.edureka.SampleBean"></bean>
<bean id="beforeAdvisor" class="co.edureka.MethodBeforeAdvisor"></bean>

<bean id="proxy" class="org.springframework.aop.framework.ProxyFactoryBean">
 <property name="target" ref="sampleBean"></property>

 <property name="interceptorNames">
 <list>
  <value>beforeAdvisor</value>
 </list>
</property>
</bean>
</beans>
```

This is our applicationContext.xml file where we have declared beans.

Note the bean declaration for ProxyFactoryBean class which is provided by Spring where we declare target bean and interceptor bean

Spring AOP Example (Contd.)



```java
package co.edureka;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;


public class TestClass {
@SuppressWarnings("resource")
public static void main(String[] args) {
 ApplicationContext context=new ClassPathXmlApplicationContext("applicationContext.xml");
 SampleBean b=(SampleBean)context.getBean("proxy");
 b.actualLogicMethod();
   }

 }
```

In TestClass we are only calling the actulaLogicMethod of SampleBean class. But as we have implemented MethodBeforeAdvice interface before method will be called first then the actualLogicMethod.

→ When you run the TestClass you will get below output

→ Here you can write code that you want to execute before method execution

→ In this method actual business logic is written

Spring – Transaction Management & JDBC

Spring is integrated with JDBC.

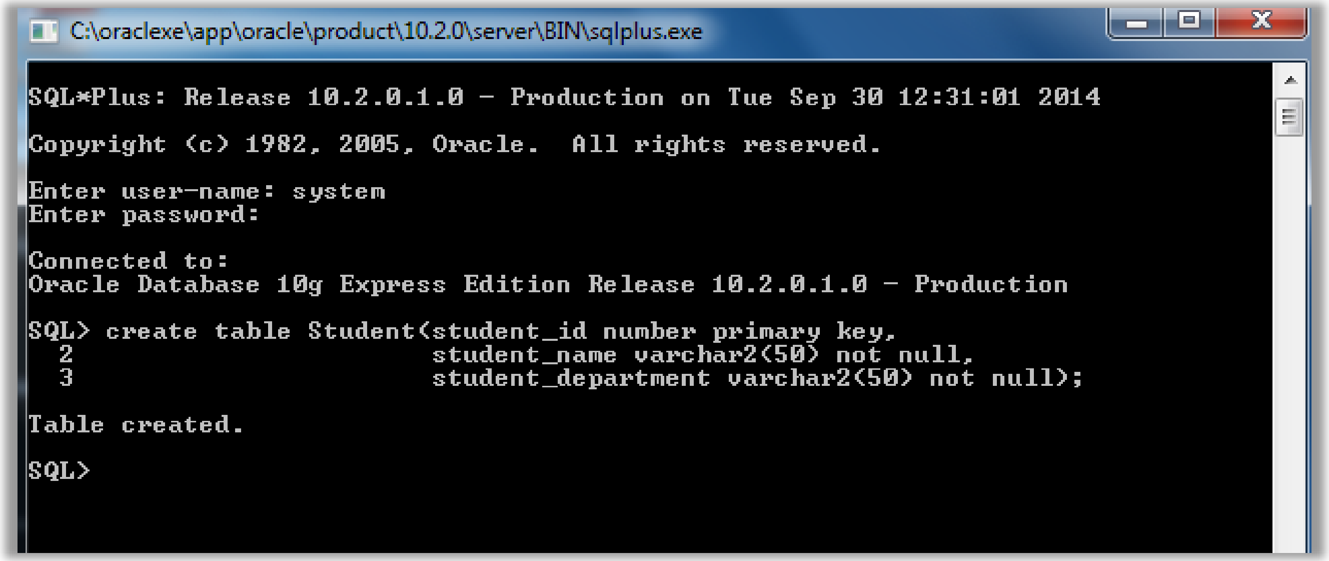
JDBCTemplate class is used to perform JDBC Operations in spring.

Database related information like connection string, user ID, password etc., will be specified in the configuration xml file.

Spring container will read this data source and will set to the class:

org.springframework.jdbc.datasource.DriverManagerDataSource, create an instance of the class and will send it to set the datasource as Dependency Injection which uses this data source for JDBC Operation.

Spring – JDBC Example

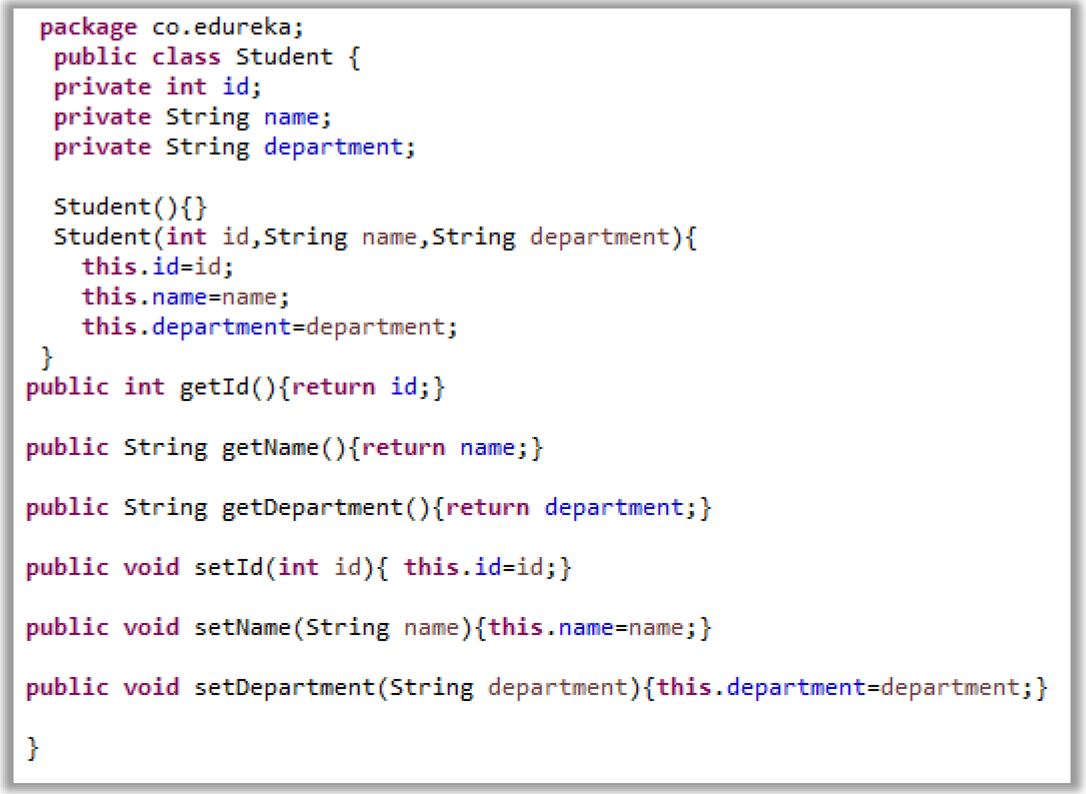

C:\oraclexe\app\oracle\product\10.2.0\server\BIN\sqlplus.exe

```
SQL*Plus: Release 10.2.0.1.0 - Production on Tue Sep 30 12:31:01 2014

Copyright (c) 1982, 2005, Oracle.  All rights reserved.

Enter user-name: system
Enter password:

Connected to:
Oracle Database 10g Express Edition Release 10.2.0.1.0 - Production

SQL> create table Student(student_id number primary key,
  2                       student_name varchar2(50) not null,
  3                       student_department varchar2(50) not null);

Table created.

SQL>
```

Create a table "student" in your database. Student table have three columns student_id,student_name and student_department

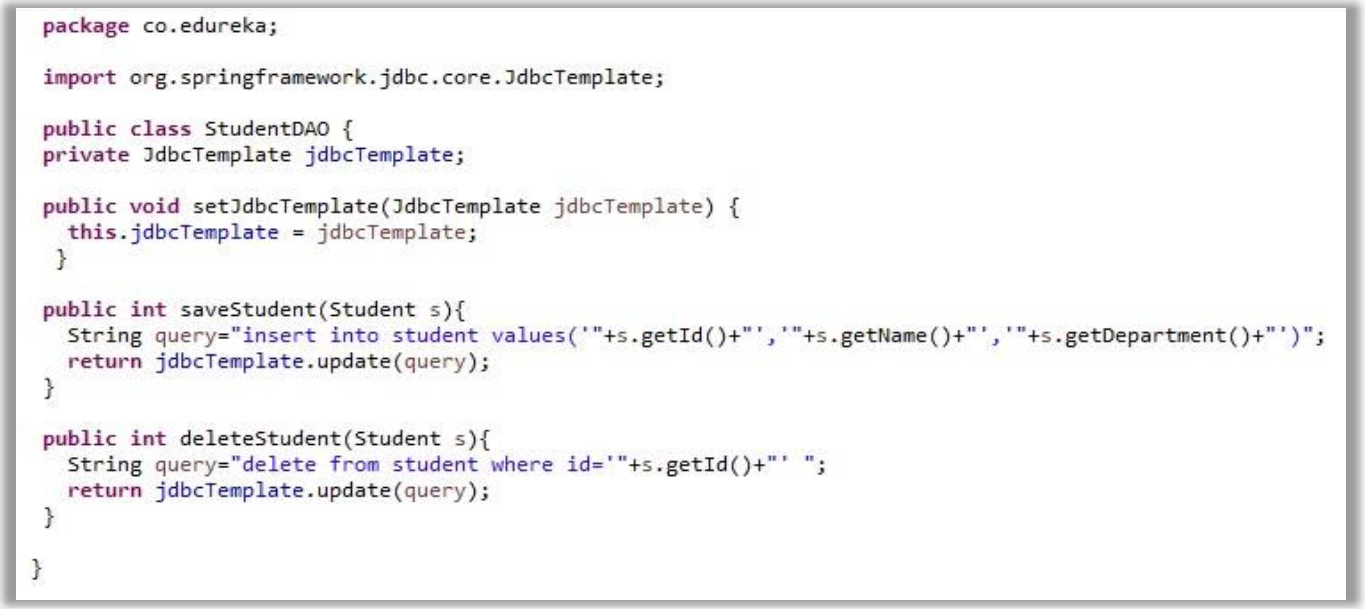
Spring JDBC Example (Contd.)

Student class



```
 package co.edureka;
  public class Student {
  private int id;
  private String name;
  private String department;

  Student(){}
  Student(int id,String name,String department){
    this.id=id;
    this.name=name;
    this.department=department;
 }
public int getId(){return id;}

public String getName(){return name;}

public String getDepartment(){return department;}

public void setId(int id){ this.id=id;}

public void setName(String name){this.name=name;}

public void setDepartment(String department){this.department=department;}

}
```

Spring JDBC Example (Contd.)

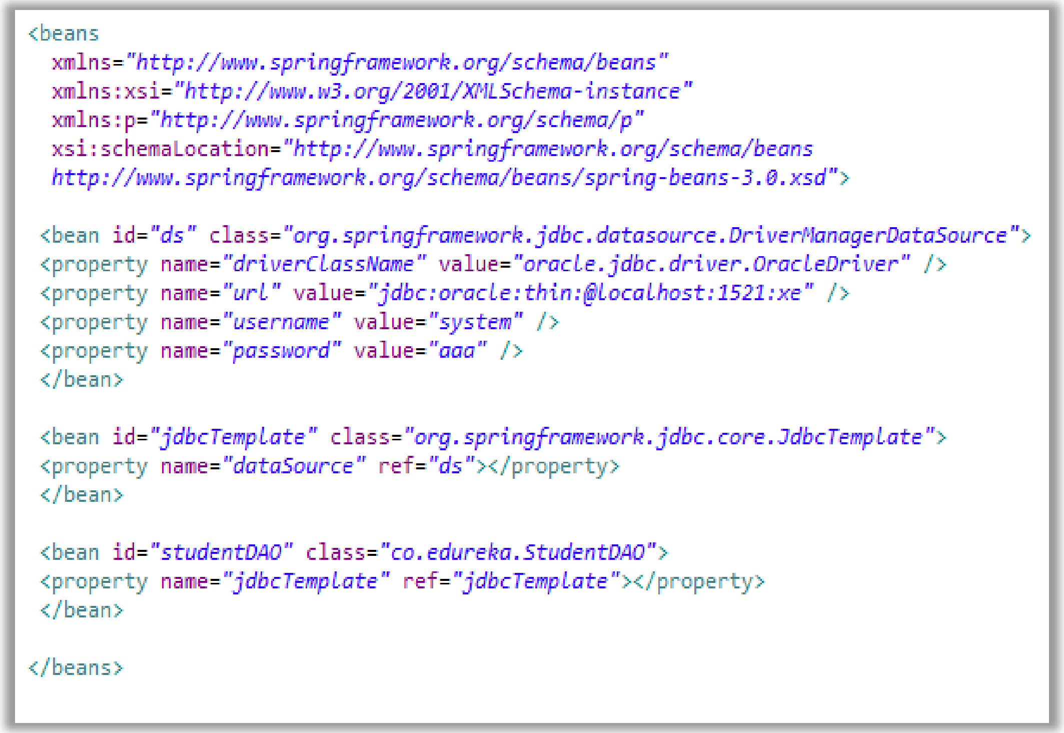


```
package co.edureka;

import org.springframework.jdbc.core.JdbcTemplate;

public class StudentDAO {
private JdbcTemplate jdbcTemplate;

public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
  this.jdbcTemplate = jdbcTemplate;
 }

public int saveStudent(Student s){
  String query="insert into student values('"+s.getId()+"','"+s.getName()+"','"+s.getDepartment()+"')";
  return jdbcTemplate.update(query);
}

public int deleteStudent(Student s){
  String query="delete from student where id='"+s.getId()+"' ";
  return jdbcTemplate.update(query);
}

}
```

We set the JDBCTamplate property with DriverManagerDataSource bean in xml file StudentDAO class which have methods to insert or delete a student. You can make as many methods as you like one for update, one to compare whether two students have same name or not etc.

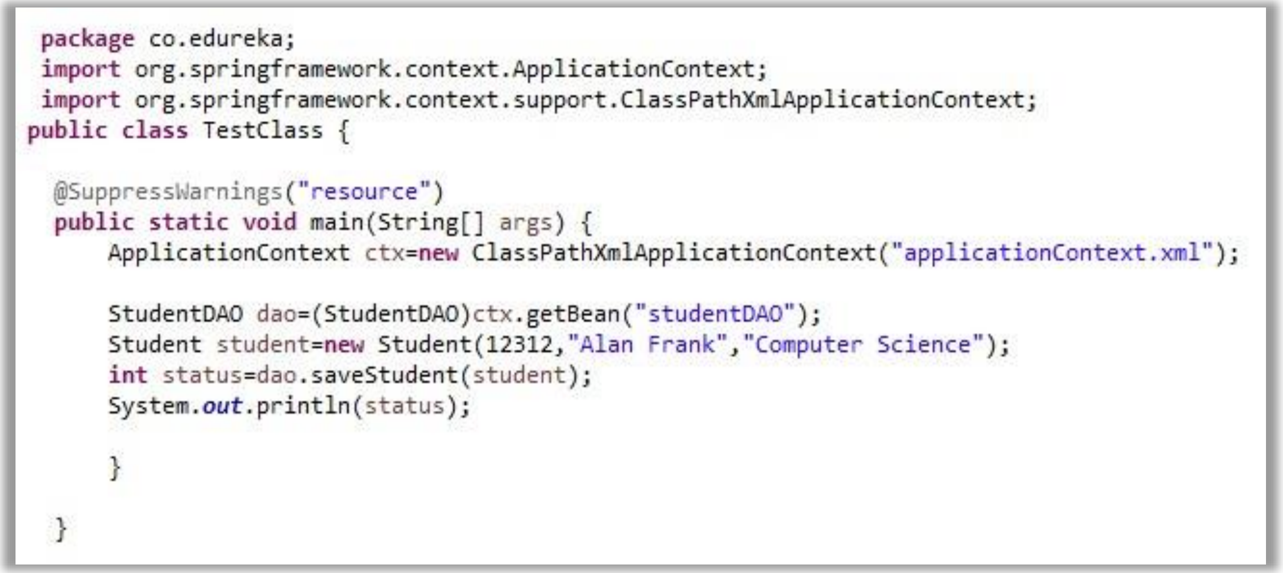
Spring JDBC Example (Contd.)

applicationContext.xml



```
<beans
  xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:p="http://www.springframework.org/schema/p"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

 <bean id="ds" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
 <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver" />
 <property name="url" value="jdbc:oracle:thin:@localhost:1521:xe" />
 <property name="username" value="system" />
 <property name="password" value="aaa" />
 </bean>

 <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
 <property name="dataSource" ref="ds"></property>
 </bean>

 <bean id="studentDAO" class="co.edureka.StudentDAO">
 <property name="jdbcTemplate" ref="jdbcTemplate"></property>
 </bean>

</beans>
```

Spring JDBC Example (Contd.)

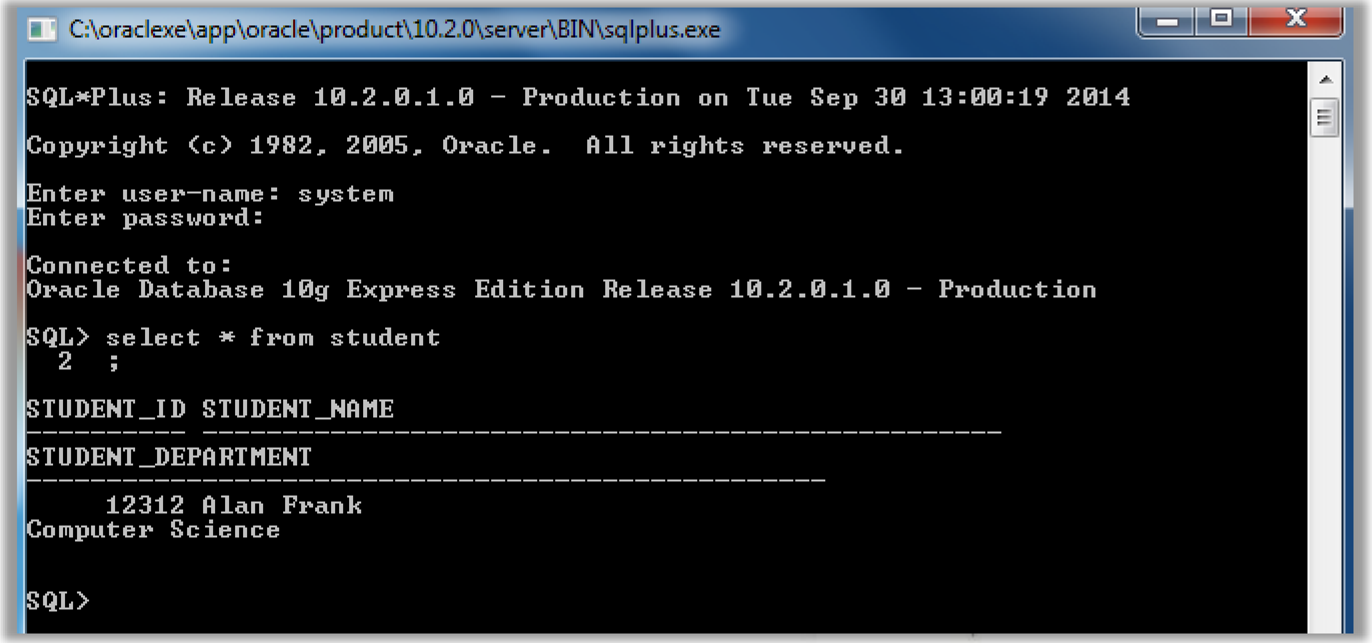


```
 package co.edureka;
 import org.springframework.context.ApplicationContext;
 import org.springframework.context.support.ClassPathXmlApplicationContext;
public class TestClass {

  @SuppressWarnings("resource")
  public static void main(String[] args) {
      ApplicationContext ctx=new ClassPathXmlApplicationContext("applicationContext.xml");

      StudentDAO dao=(StudentDAO)ctx.getBean("studentDAO");
      Student student=new Student(12312,"Alan Frank","Computer Science");
      int status=dao.saveStudent(student);
      System.out.println(status);

      }

  }
```

In the TestClass we are getting the studentDAO bean which we have declared in the xml file. Then we create a Student class object and pass that object to saveStudent method. So after executing the TestClass that Student will be added in to database.

Spring JDBC Example(contd.)

Lets check the output:


C:\oraclexe\app\oracle\product\10.2.0\server\BIN\sqlplus.exe

```
SQL*Plus: Release 10.2.0.1.0 - Production on Tue Sep 30 13:00:19 2014

Copyright (c) 1982, 2005, Oracle.  All rights reserved.

Enter user-name: system
Enter password:

Connected to:
Oracle Database 10g Express Edition Release 10.2.0.1.0 - Production

SQL> select * from student
  2  ;

STUDENT_ID STUDENT_NAME
---------- --------------------------------------------------
STUDENT_DEPARTMENT
--------------------------------------------------
     12312 Alan Frank
Computer Science


SQL>
```

Spring – Transaction Management

Sequence of transactions with database can be treated as single unit of work.

Transactions should follow **ACID** properties:

Atomicity – If the set of transactions is successful then all the entries should be part of the database. If there is failure in some part of the transactions then entire set should be reverted back.

Consistency – Consistency means that you guarantee that your data will be consistent; none of the constraints you have on related data will ever be violated like primary key, foreign key etc.

Isolation – Even though multiple transactions are performed at the same time, each transaction should be isolated without corrupting the data.

Durability – Should be available in long term even if there is a power failure etc.

Spring – Transaction Management (Contd.)

TransactionDefinition class is used to define the transaction properties like isolation etc.

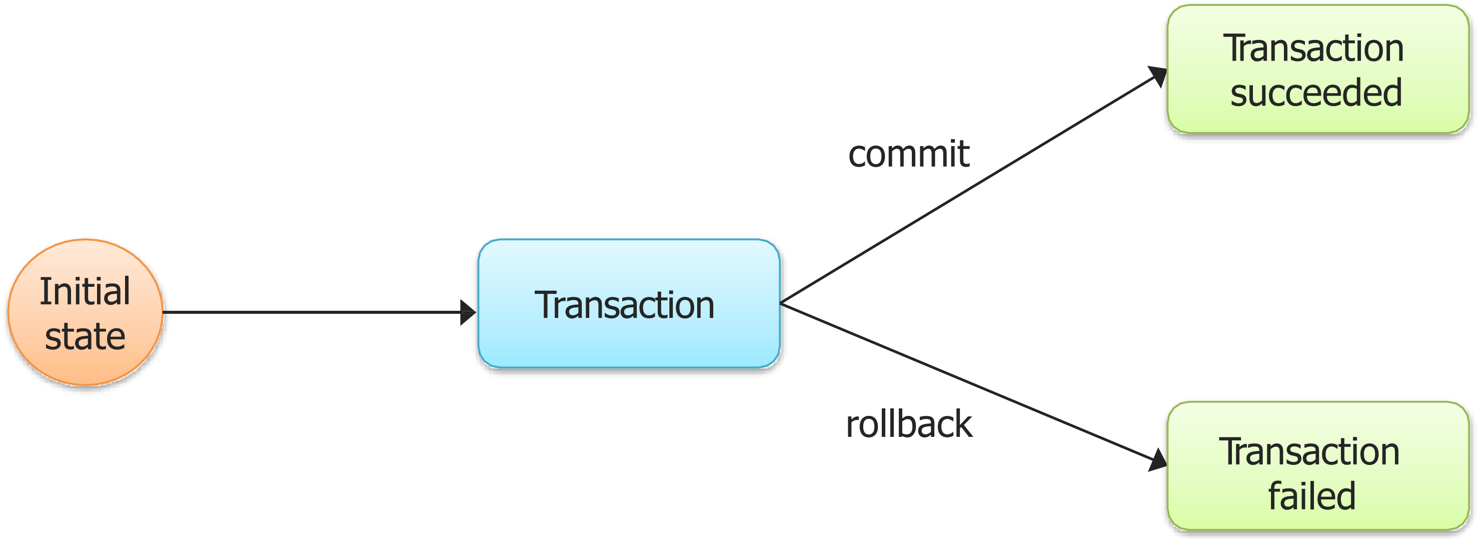
PlatformTransactionManager class creates a transaction object either new or the current one and returns this as Transaction Status.

```
PlatformTransactionManager transactionManager;
TransactionStatus status = transactionManager.getTransaction(def);
```

```
transactionManager.commit(status);
```

```
transactionManager.rollback(status);
```

Spring – Transaction Management Flow



Spring – Integration with Hibernate

Hibernate needs the following files to perform the db operations:

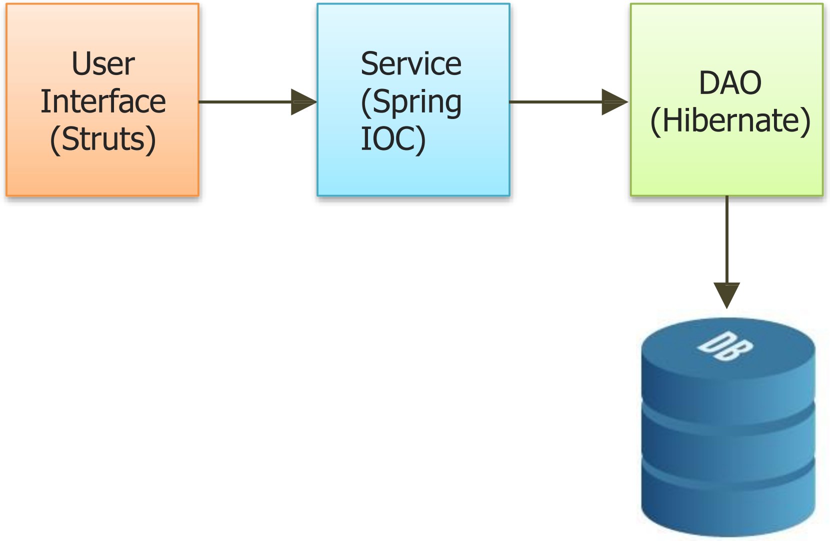
» Bean class → which is the main object.

» Hbm file → Which maps between bean object and db table and its columns.

» Client file → Which uses the bean and performs the db transaction.

» DAO file → This is optional. If DAO is written then client file have to access DAO to perform db transactions.

Hibernate Integration



Spring provides HibernateTemplate class for hibernate operations.

HibernateTemplate provides methods for inserting, deleting, updating and querying the data.

Some of the methods are:

→ Save()
→ Delete()
→ Find()
→ update()

Hibernate Integration (Contd.)

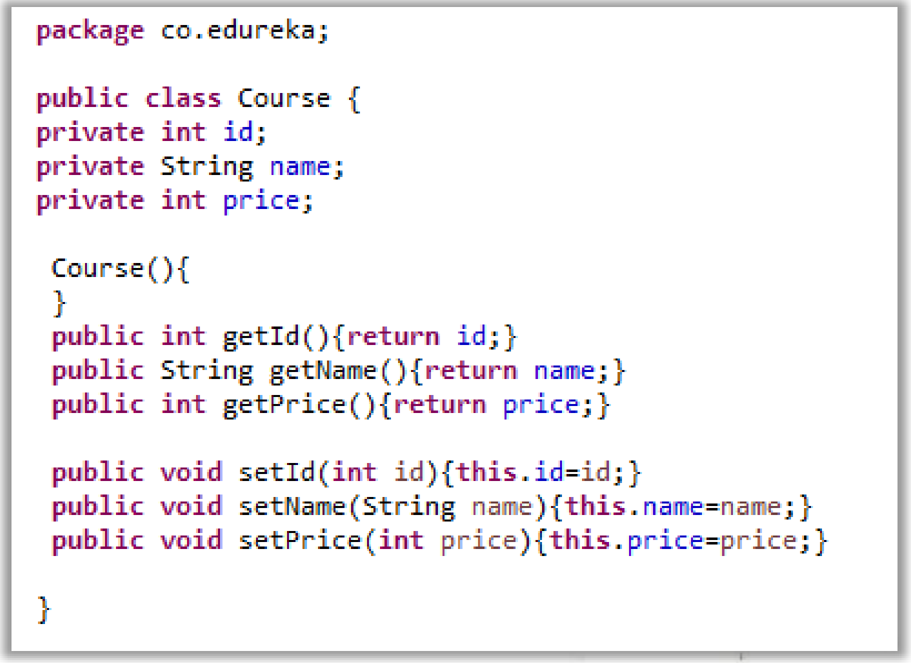
→ Provide all the hibernate integration parameters as part of configuration xml file.

→ This data can be passed to bean/dao by setter based dependency injection or constructor based dependency injection.

Lets understand it with an example…..

Spring and Hibernate Example

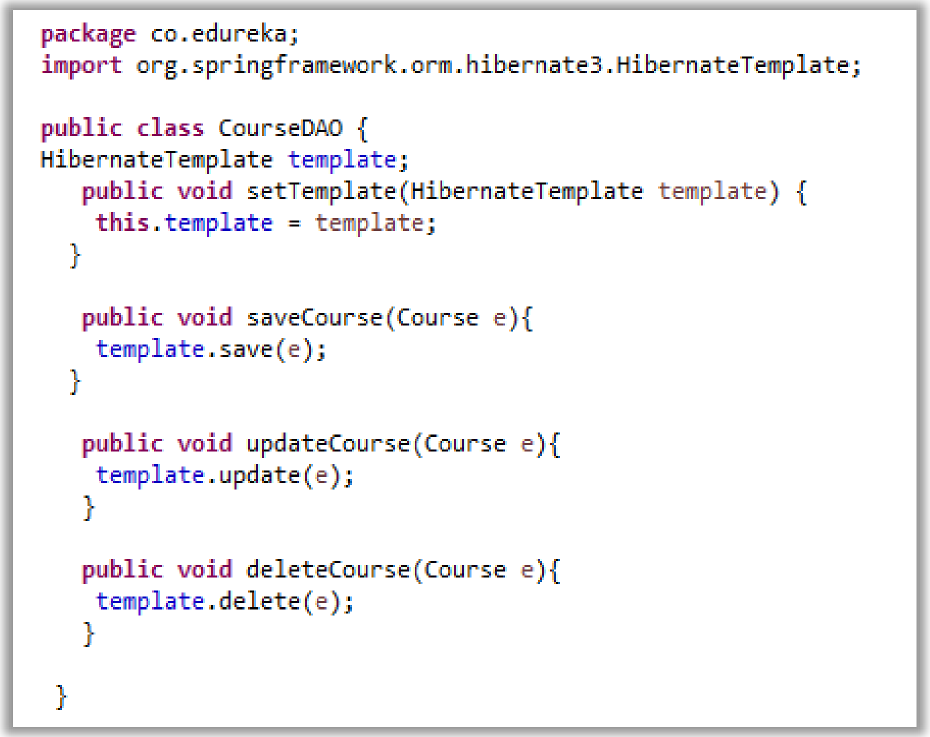
Course class



```
package co.edureka;

public class Course {
private int id;
private String name;
private int price;

 Course(){
 }
 public int getId(){return id;}
 public String getName(){return name;}
 public int getPrice(){return price;}

 public void setId(int id){this.id=id;}
 public void setName(String name){this.name=name;}
 public void setPrice(int price){this.price=price;}

}
```

Spring and Hibernate Example (Contd.)

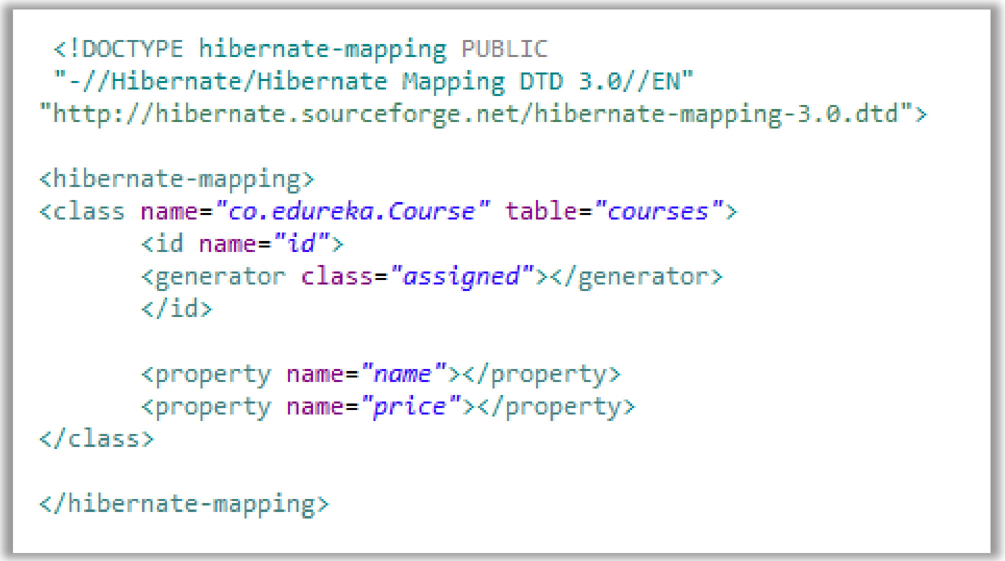
CourseDAO class



```
package co.edureka;
import org.springframework.orm.hibernate3.HibernateTemplate;

public class CourseDAO {
HibernateTemplate template;
   public void setTemplate(HibernateTemplate template) {
    this.template = template;
  }

   public void saveCourse(Course e){
    template.save(e);
  }

   public void updateCourse(Course e){
    template.update(e);
   }

   public void deleteCourse(Course e){
    template.delete(e);
   }

 }
```

Spring and Hibernate Example (Contd.)

course.hbm.xml

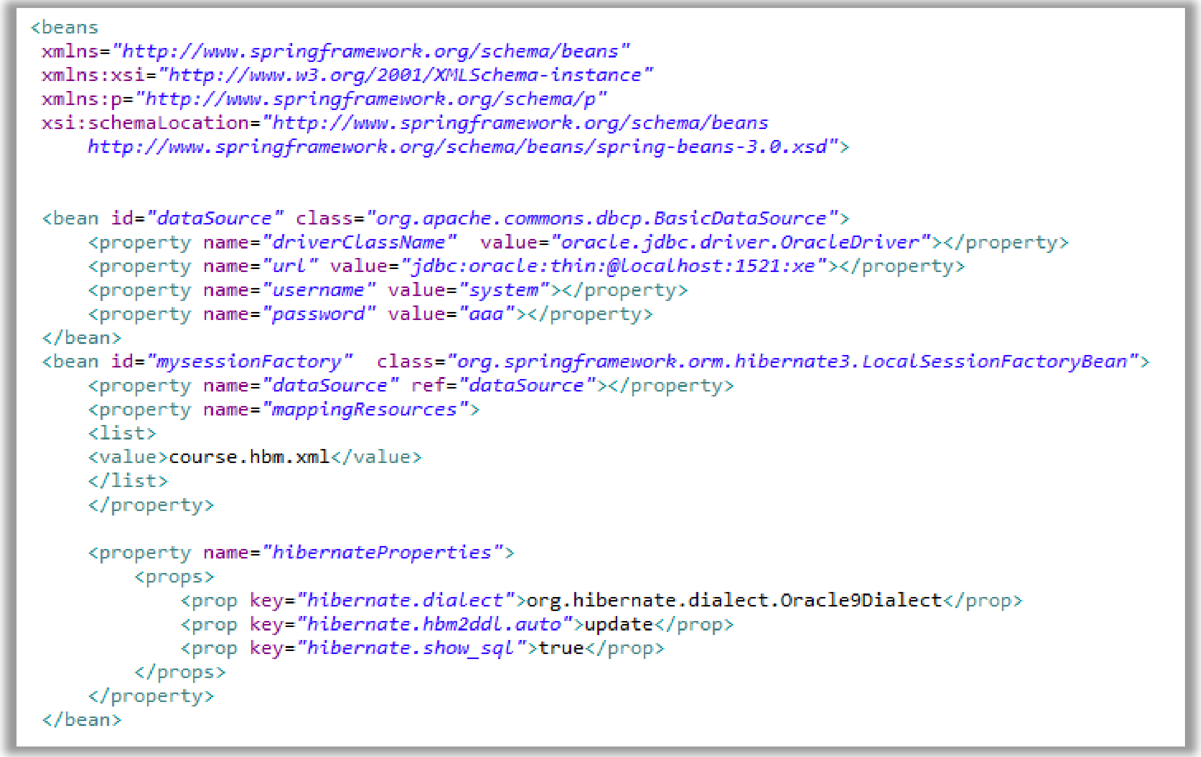


```
 <!DOCTYPE hibernate-mapping PUBLIC
 "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
<class name="co.edureka.Course" table="courses">
       <id name="id">
       <generator class="assigned"></generator>
       </id>

       <property name="name"></property>
       <property name="price"></property>
</class>

</hibernate-mapping>
```

This is the mapping file which will take care of mapping between your Courses table and Course class

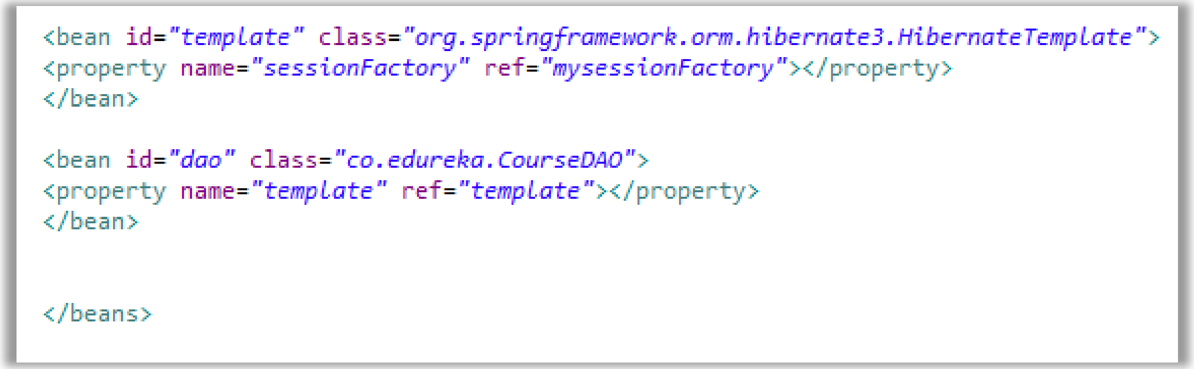
Spring and Hibernate Example (Contd.)

applicationContext.xml (contd.)



```
<beans
 xmlns="http://www.springframework.org/schema/beans"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:p="http://www.springframework.org/schema/p"
 xsi:schemaLocation="http://www.springframework.org/schema/beans
     http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">


 <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">
     <property name="driverClassName"  value="oracle.jdbc.driver.OracleDriver"></property>
     <property name="url" value="jdbc:oracle:thin:@localhost:1521:xe"></property>
     <property name="username" value="system"></property>
     <property name="password" value="aaa"></property>
 </bean>
 <bean id="mysessionFactory"  class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
     <property name="dataSource" ref="dataSource"></property>
     <property name="mappingResources">
     <list>
     <value>course.hbm.xml</value>
     </list>
     </property>

     <property name="hibernateProperties">
         <props>
             <prop key="hibernate.dialect">org.hibernate.dialect.Oracle9Dialect</prop>
             <prop key="hibernate.hbm2ddl.auto">update</prop>
             <prop key="hibernate.show_sql">true</prop>
         </props>
     </property>
 </bean>
```

Spring and Hibernate Example (Contd.)

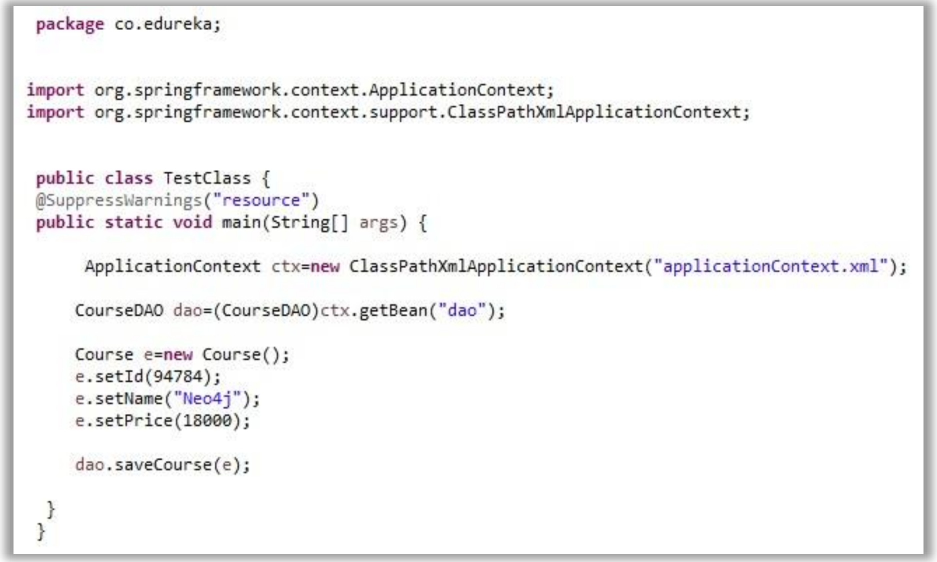


```
<bean id="template" class="org.springframework.orm.hibernate3.HibernateTemplate">
<property name="sessionFactory" ref="mysessionFactory"></property>
</bean>

<bean id="dao" class="co.edureka.CourseDAO">
<property name="template" ref="template"></property>
</bean>


</beans>
```

In the applicationContext.xml file we are declaring lots of bean definition.
One is for DataSource definition, one is for Hibernate Session Factory with the mapping resource course.hbm.xml.

Then we are creating HibernateTemplate bean setting its sessionFactory property to mysessionfactory.
Last is the bean definition of CourseDAO class setting its template property to HibernateTemplate bean.

Spring and Hibernate Example (Contd.)

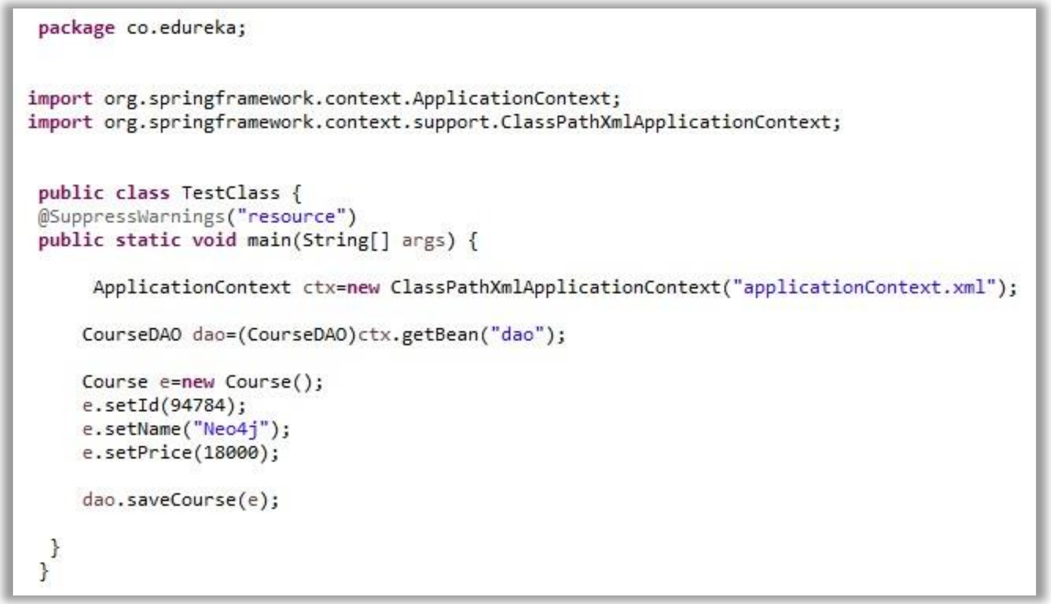


```
 package co.edureka;


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


 public class TestClass {
 @SuppressWarnings("resource")
 public static void main(String[] args) {

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

     CourseDAO dao=(CourseDAO)ctx.getBean("dao");

     Course e=new Course();
     e.setId(94784);
     e.setName("Neo4j");
     e.setPrice(18000);

     dao.saveCourse(e);

  }
 }
```

When you run the TestClass, you will see one row in courses table.

Note to run this program make sure you have required jars, if not download them and add in your project's build path.

Spring and Hibernate Example (Contd.)

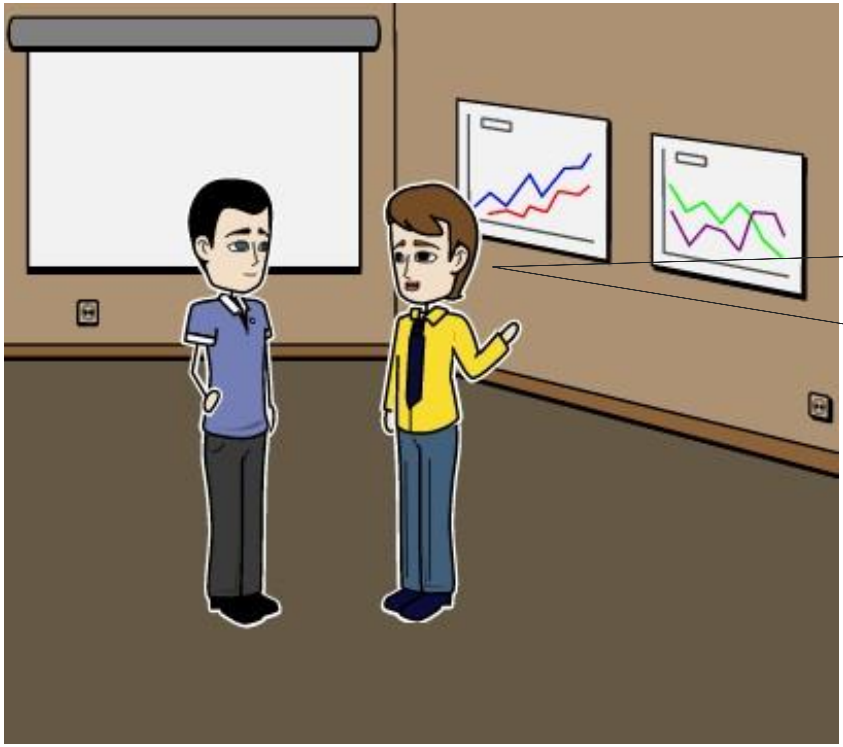


```
 package co.edureka;


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


 public class TestClass {
 @SuppressWarnings("resource")
 public static void main(String[] args) {

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

     CourseDAO dao=(CourseDAO)ctx.getBean("dao");

     Course e=new Course();
     e.setId(94784);
     e.setName("Neo4j");
     e.setPrice(18000);

     dao.saveCourse(e);

  }
 }
```

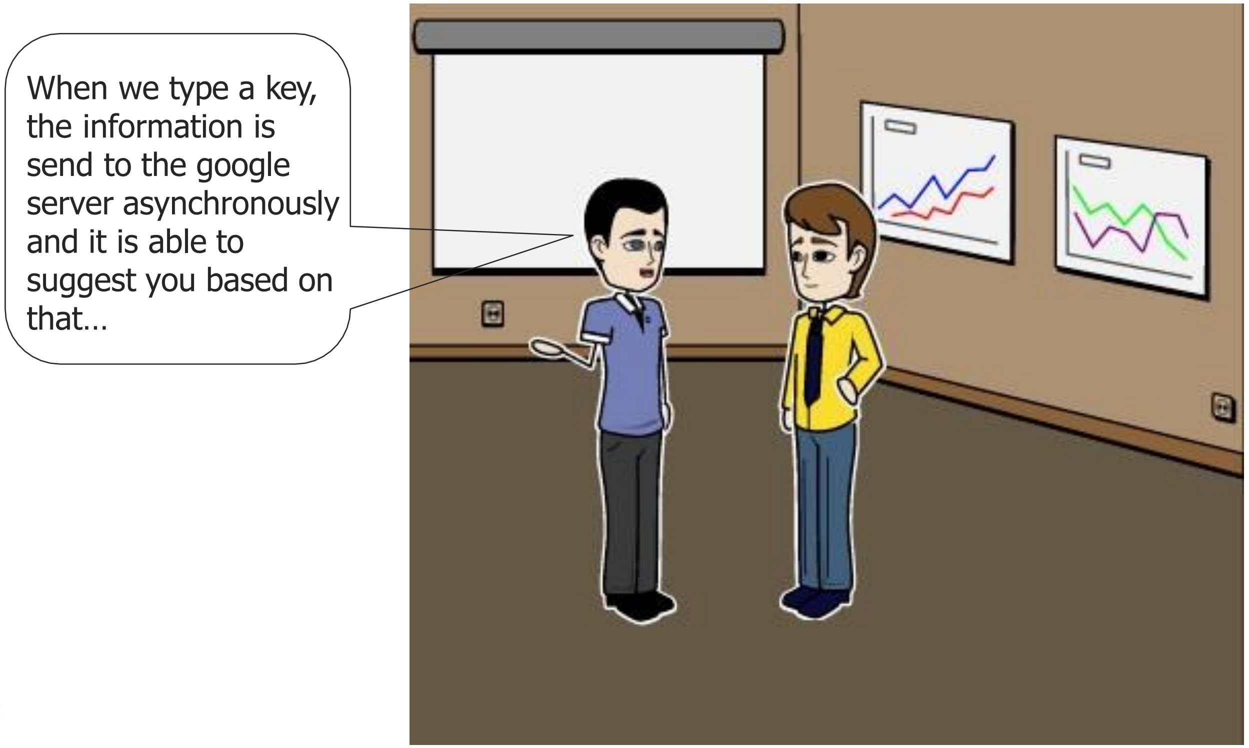
When you run the TestClass, you will see one row in courses table.

Note to run this program make sure you have required jars, if not download them and add in your project’s build path.

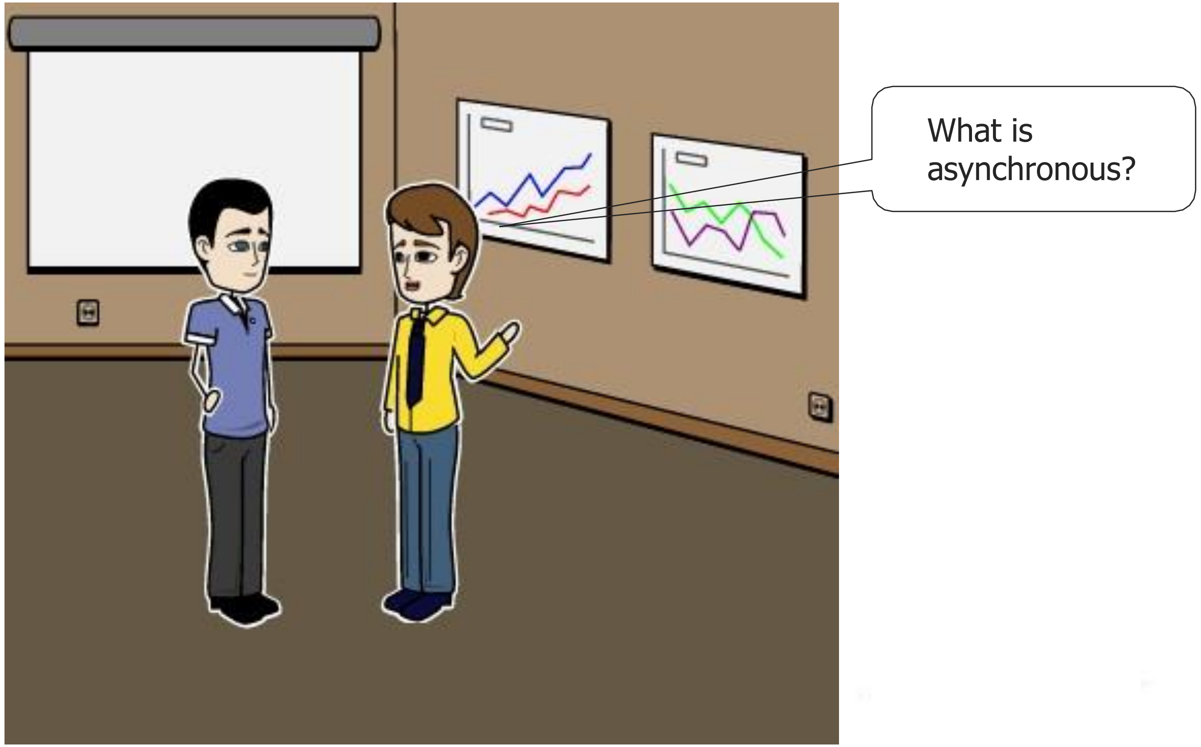
Mark has a Doubt


John, when we search for something in google, we get lot of suggestions. How does this happen?

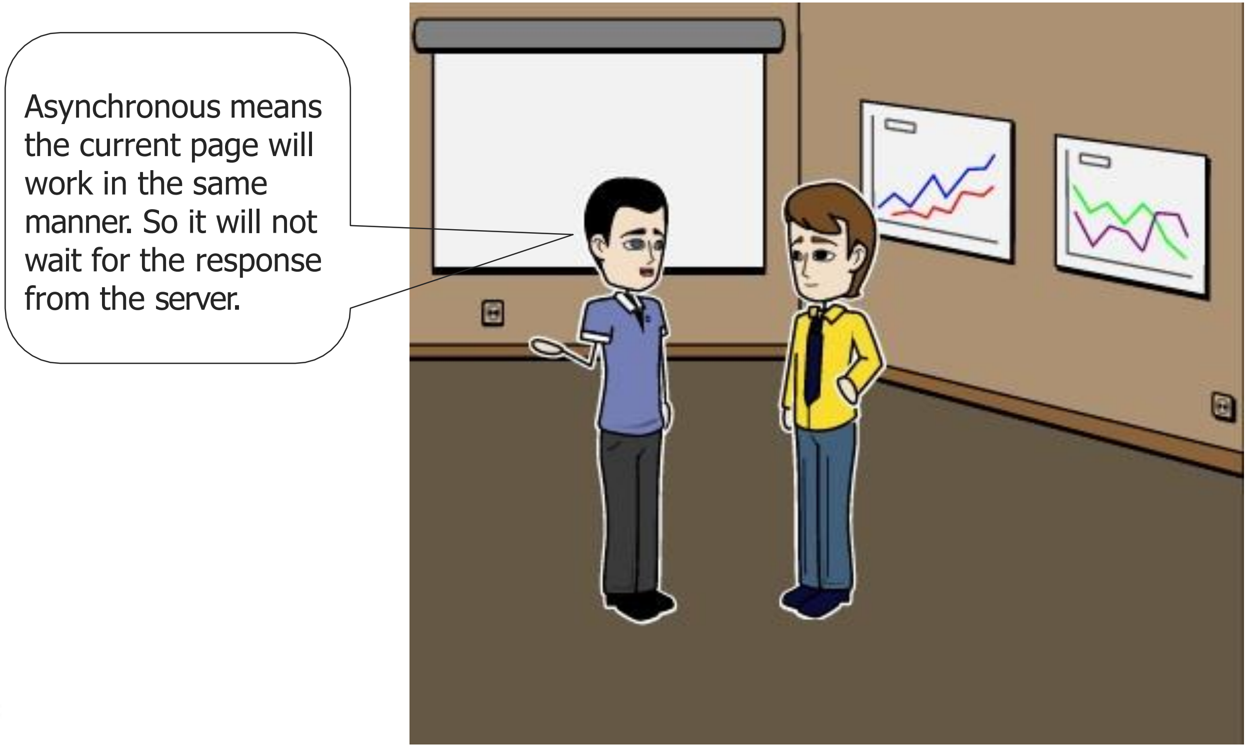
John helps Mark..


When we type a key, the information is send to the google server asynchronously and it is able to suggest you based on that…

John helps Mark..



John helps Mark..


Asynchronous means the current page will work in the same manner. So it will not wait for the response from the server.

AJAX

→ AJAX stands for Asynchronous JavaScript and XML.

→ Client communicates with server asynchronously (That is in the background).

→ HTML and CSS can be used in addition to AJAX in the same form for display the form and style of the form.

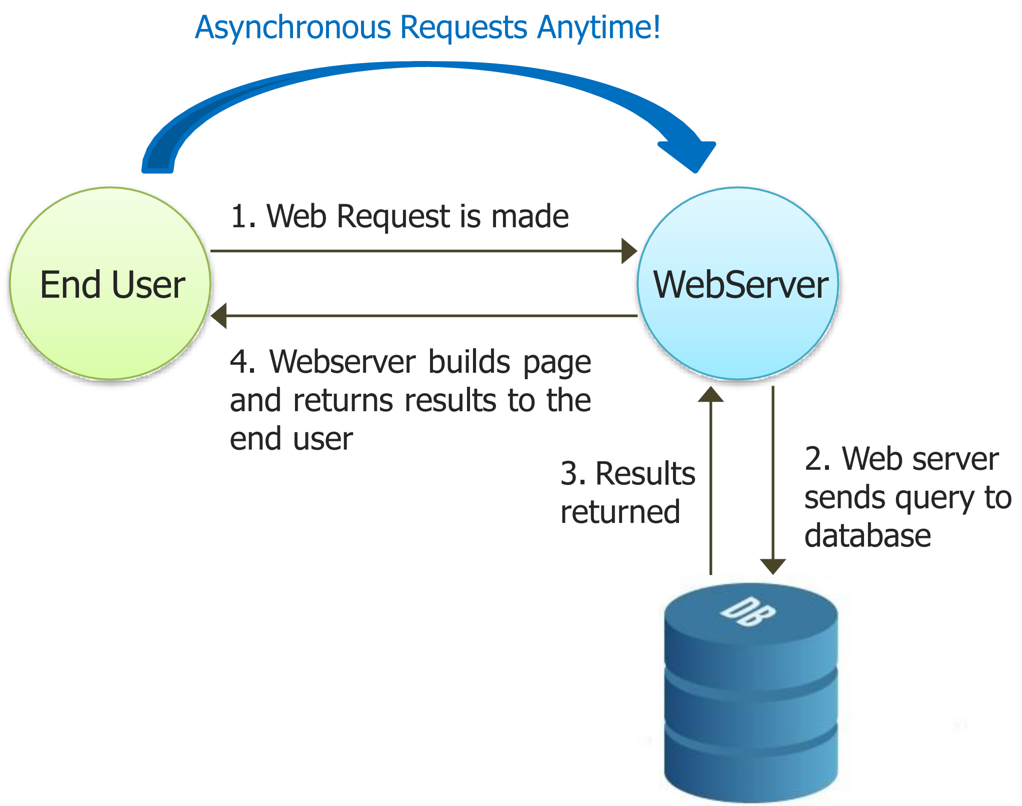
Annie's Question

Where have you seen such communication?

Annie’s Answer

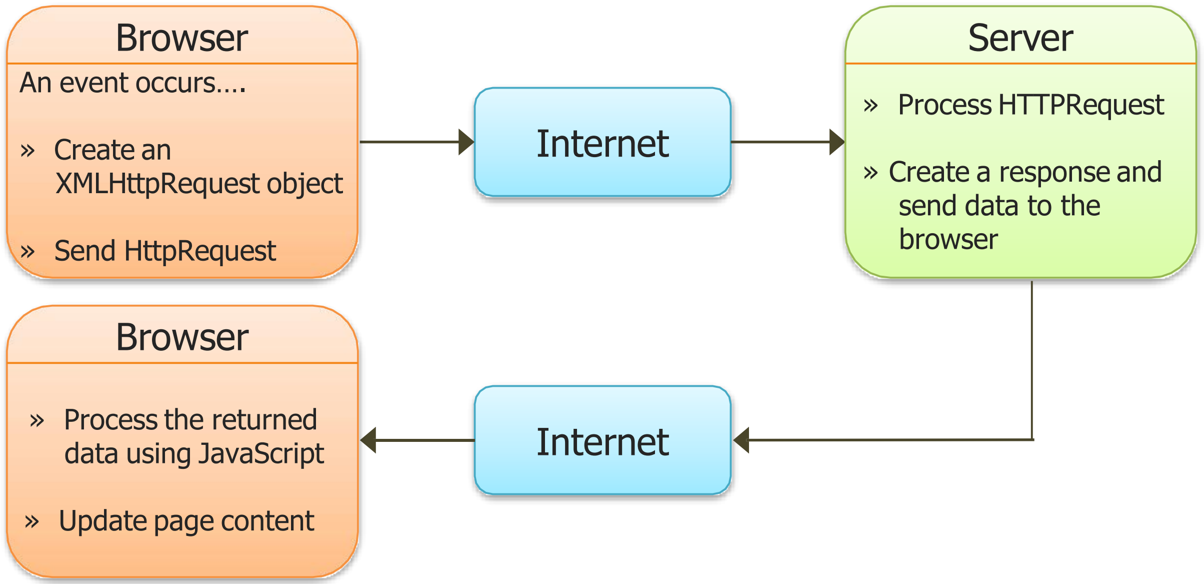
In many web forms, when the country is selected, its states are populated without form being submitted to the server.

Ajax Architecture



Asynchronous Requests Anytime!
End User
WebServer
1. Web Request is made
4. Webserver builds page and returns results to the end user
3. Results returned
2. Web server sends query to database
DB

How Ajax works?


Browser
An event occurs....
» Create an XMLHttpRequest object
» Send HttpRequest
Internet
Server
» Process HTTPRequest
» Create a response and send data to the browser
Internet
Browser
» Process the returned data using JavaScript
» Update page content

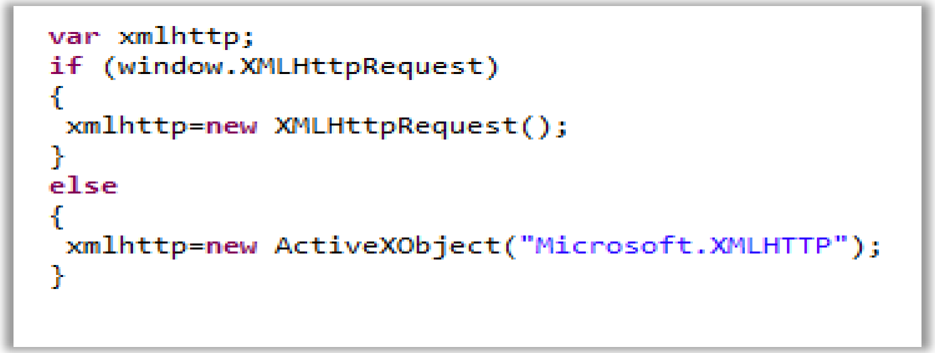
Ajax Request Object

Through Request object only request has to be sent to the server.

For IE5 and IE6 `request=new ActiveXObject("Microsoft.XMLHTTP");` will be used to get the request object.

For IE7 and other browsers `request=new XMLHttpRequest();` will be used to get the request object.

Ajax – Request Object



```
var xmlhttp;
if (window.XMLHttpRequest)
{
 xmlhttp=new XMLHttpRequest();
}
else
{
 xmlhttp=new ActiveXObject("Microsoft.XMLHTTP");
}
```

A variable request is declared.

If the browser is IE5/IE6 then window.ActiveXObject condition is true and request object is obtained from Microsoft.XMLHTTP

If the browser is IE7, Chrome, Safari, Firefox etc., then request object is obtained from XMLHttpRequest object

Ajax – Request Methods

To send the request to the server the following two methods are used:

(1) open()
(2) send()

1. open (GET/POST, url of the location, true/false for asynchronous/synchronous communication).

2. send() → sends the request to the specified url in the request object using GET/POST in asynchronous or synchronous mode.

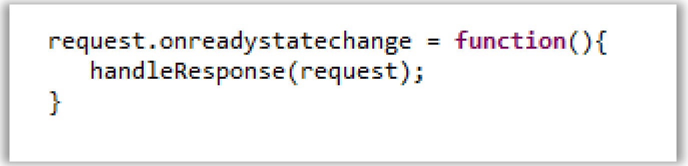
Ajax – Request Events

XMLHttpRequest object have readyState property which defined the current state of XMLHttpRequest object readyState property can have following possible values

1 – Request is not initialized
2– Server connection established
2 – Request sent
3 – Processing request
4 – Processing is done and response is ready

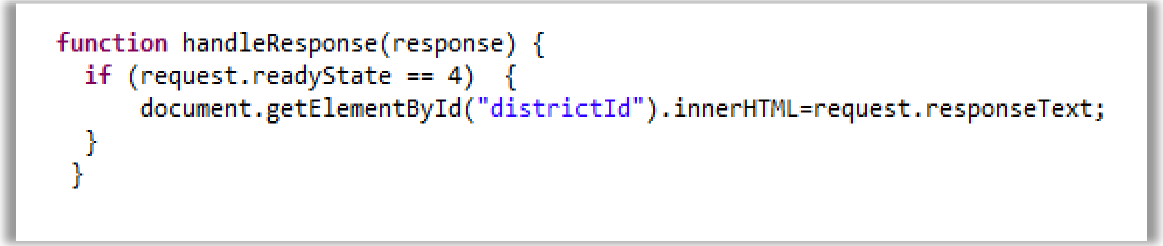
Whenever state of the request is changed, onreadystatechange() function is called. In this function, user can check for status 4 and perform the necessary action with the response of the server.

Ajax – Response Handling



```
request.onreadystatechange = function(){
   handleResponse(request);
}
```

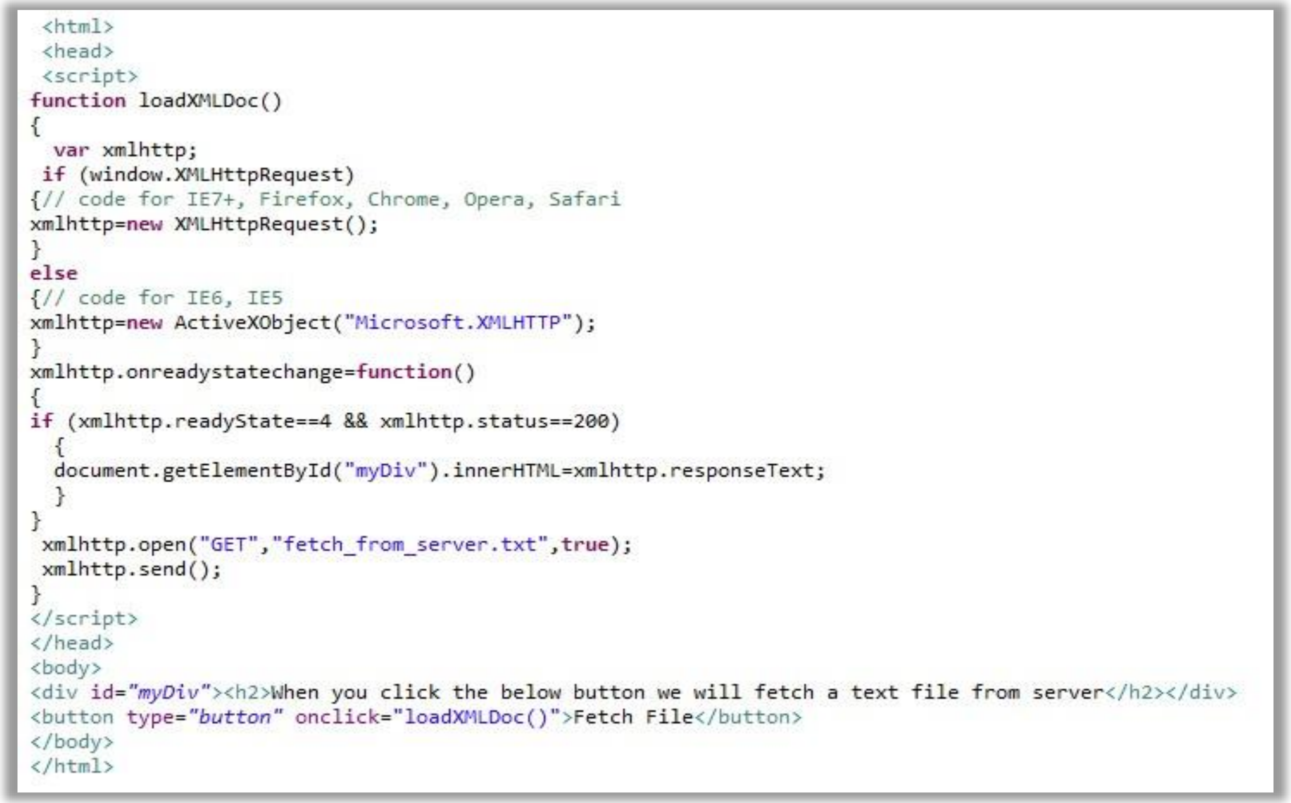
Here onreadystatechange is called whenever there is change in the state of the request object, handleResponse() is executed. Code for this function is given below:



```
function handleResponse(response) {
  if (request.readyState == 4)  {
      document.getElementById("districtId").innerHTML=request.responseText;
  }
 }
```

This function checks for state equals 4 and changes the districtId text to the response given by the server.

Ajax Example



```
 <html>
 <head>
 <script>
function loadXMLDoc()
{
  var xmlhttp;
 if (window.XMLHttpRequest)
{// code for IE7+, Firefox, Chrome, Opera, Safari
xmlhttp=new XMLHttpRequest();
}
else
{// code for IE6, IE5
xmlhttp=new ActiveXObject("Microsoft.XMLHTTP");
}
xmlhttp.onreadystatechange=function()
{
if (xmlhttp.readyState==4 && xmlhttp.status==200)
  {
  document.getElementById("myDiv").innerHTML=xmlhttp.responseText;
  }
}
 xmlhttp.open("GET","fetch_from_server.txt",true);
 xmlhttp.send();
}
</script>
</head>
<body>
<div id="myDiv"><h2>When you click the below button we will fetch a text file from server</h2></div>
<button type="button" onclick="loadXMLDoc()">Fetch File</button>
</body>
</html>
```

Ajax Example (Contd.)

Lets go through the code

- We have defined loadXMLDoc function and we will call that function whenever user clicks the button.
- Now depending upon client's browser we are assigning the object to xmlhttp variable.
- In next lines we are checking the state of readyState and status property.
  If readyState is 4 and status is 200, it means server have completed his response.

  If that's the case we are setting the div tag's innerHTML to the responseText that we got from server.

  Now initially readyState will not be 4, so if condition will be false.
- Next we are calling the open method specifying the type of request, URL of file to fetch and type of request to be made synchronously or asynchronously.
- Then we are sending the request to server.

Design Patterns

Design patterns are a standard way of solving a problem.

Design patterns are used during the design of the system.

There are 2 major kinds of design patterns:

1. Java design patterns
2. J2EE design patterns

Java & J2EE Design Patterns

J2EE Design patterns refers to all the patterns which are related to J2EE.

There are 3 kinds of Java design patterns.

1. Creational → Patterns for the way of creating object.
2. Structural → Patterns for relating objects can be related and made larger patterns.
3. Behavioural → Patterns dealing with interaction of objects and responsibility of objects.

Creational Design Patterns

Singleton → Ensures only one object is created for the entire application.

Factory → Creates the objects without exposing the creation logic to the client.

Usage: Creating DOMFactory (DocumentBuilderFactory) or SAX Factory Object.

AbstractFactory → Provides an interface for creating families of related or dependent objects without specifying their concrete classes. For example, creating multiple types of cars using a single factory object.

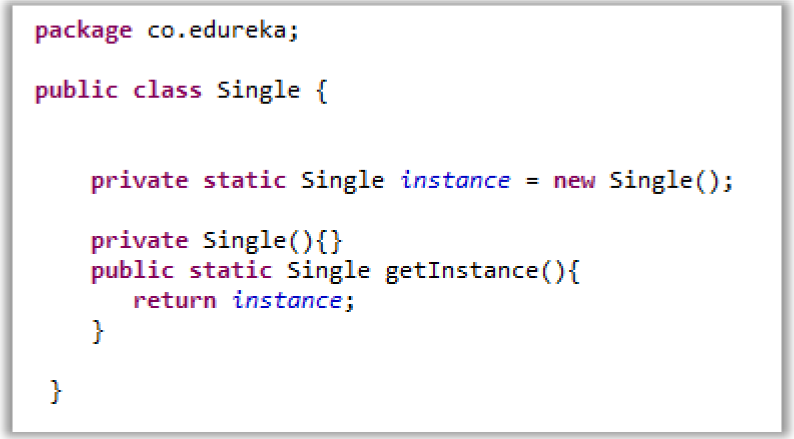
Prototype → Used to copy one object to another instead of creating a new object and filling up all the attributes.Prototype.

Builder → Separate the construction of complex object from its representation so that the same construction can create different representation. For example, meal in mc Donald's have to make build burger, build coke, build finger chips and deliver...

Object Pool → Reuses and shares the objects. Application can implement to ensure maximum number of db connections.

Singleton Pattern Example

Sometimes we want to restrict that , only one instance of a class to be made.



```java
package co.edureka;

public class Single {


    private static Single instance = new Single();

    private Single(){}
    public static Single getInstance(){
       return instance;
    }

 }
```

In the above class we have implemented Singleton pattern for class Single.

Note that you can not instantiate that class from outside as the constructor is marked private.
Now to get an instance of Single class call the getInstance static method of Single class.

Structural Design Patterns

Adapter → Converts one interface to another that the client wants.

Bridge → De-couple the abstraction from its implementation so that two can vary independently. Bridge the functionality between the type and its dependency.

Composite → Allow clients to operate in generic manner on objects that may or may not represent hierarchy of objects.

Decorator → To extend or modify the behaviour of an instance at runtime. A wrapper object can be constructed around an object to extend the functionality. Wrapper will do its job before/after the object behaviour.

Facade → Hides the complexities of the system and provides an interface to the client for client to access the system. To start a car, we need not control the engine and valves and all. Just insert the key in keyhole and turn it so that the car starts.

Flyweight → This pattern helps to improve the performance of the system by sharing the objects instead of creating the objects every time.

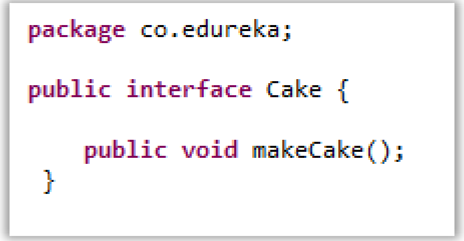
Momento → Used to store the state of the object. This pattern is developed for Ctrl + z(Undo) and Ctrl + y(Redo).

Proxy → A class represents another class as proxy and can execute the methods of another class too.

Decorator Pattern Example

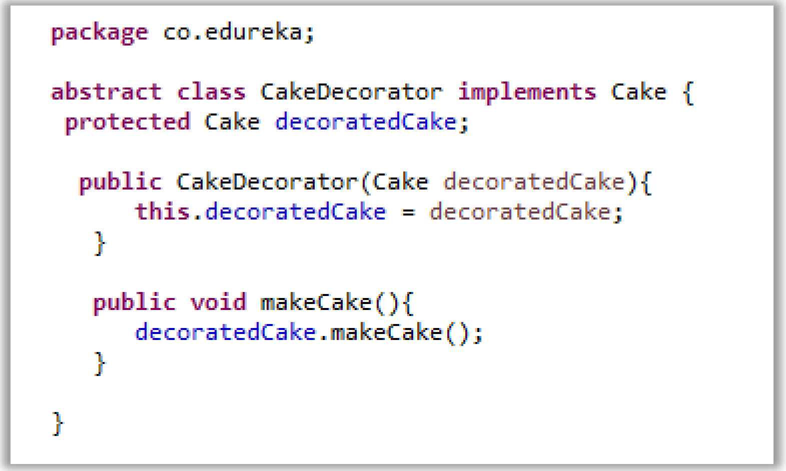
Decorator pattern is used when we want to extend the already available functionality.

So In decorator pattern we use the functionality of the already available class and add extra code that we want to perform on top of that.



```java
package co.edureka;

public interface Cake {

    public void makeCake();
 }
```

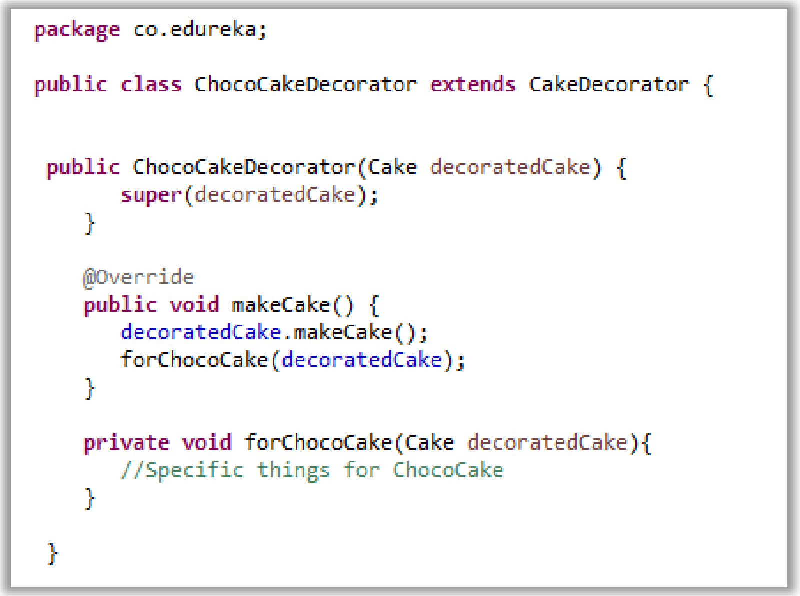
Cake Interface



```java
package co.edureka;

abstract class CakeDecorator implements Cake {
 protected Cake decoratedCake;

  public CakeDecorator(Cake decoratedCake){
      this.decoratedCake = decoratedCake;
   }

   public void makeCake(){
      decoratedCake.makeCake();
   }

}
```

CakeDecorator Class

Decorator Pattern Example (Contd.)



```
package co.edureka;

public class ChocoCakeDecorator extends CakeDecorator {


 public ChocoCakeDecorator(Cake decoratedCake) {
        super(decoratedCake);
    }

    @Override
    public void makeCake() {
        decoratedCake.makeCake();
        forChocoCake(decoratedCake);
    }

    private void forChocoCake(Cake decoratedCake){
        //Specific things for ChocoCake
    }

 }
```

This is the concrete class ChocoCakeDecorator.

Note that for makeCake it calls the base class implementation and then it calls forChocoCake method because it is specific for Choco cake

Behavioural Design Patterns

Chain of responsibility → Avoiding the direct coupling between sender and receiver by allowing some more objects to handle the request. Chain the receiving objects and pass the request along the chain until the object which can handle the request.

Command → A command is wrapped inside an object and passed to handle. At the receiving end client request is interpreted and handled accordingly. For example, customer places 4 items for the dinner as starters, food and coke. This goes in a single request. The food is prepared based on request and will be served.

Interpreter → Language / grammar Interpretation. For example, SQL Parsing, Parsing of a particular file format.

Iterator → Writing iterator for our objects using Iterator.

Mediator → Mediator provides many to many functionality. Defines an object that encapsulates how a set of objects should interact.

Observer → It is one to many relationship. If one object is modified, all its dependent objects are notified.

Behavioural Design Patterns (Contd.)

State → Behaviour of the object changes based on state of other object.

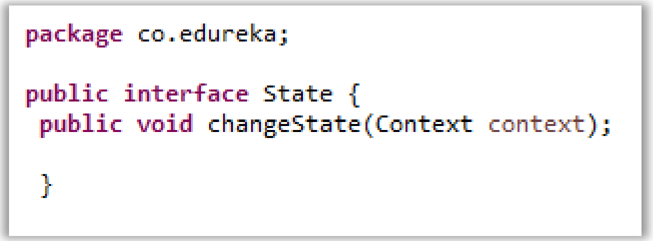
Strategy → Based on the input strategy of the algorithm is used.

Template → Common functionality is defined in a class and rest of the functionality is written into its own class. All workers will have common functions like getup(), eat(), work(), sleep() etc., for specific worker type the work will be different for plumber, carpenter, fireman, gardner, developer. Write the individual classes about their work(). Template will give the same meaning but different implementation based on input. Palindrome for string and integer. A template can be written for this.

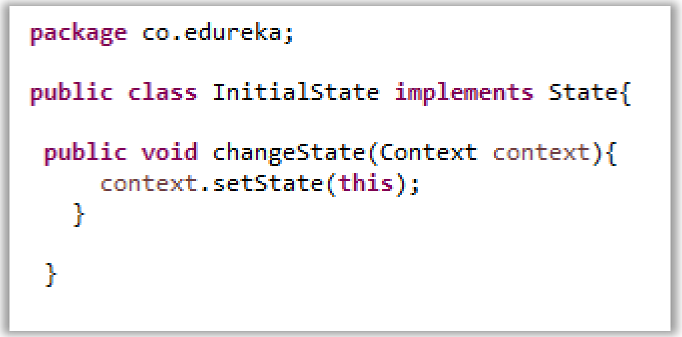
Visitor → As the visitor changes the behaviour of the object/system changes. For example, in shopping cart we can have many items. Vegetables separate pack and books and other in separate pack. Another example, based on the input type the system takes the input that way. Input devices can be keyboard, mouse, light pen etc.

State Pattern Example

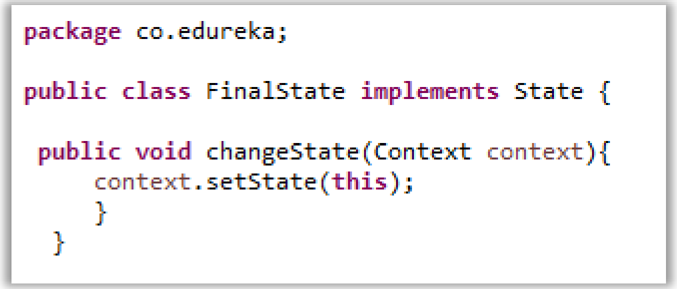
When the state of an object depends upon state of another object we can implement that scenario using State pattern. Lets understand it with an example.



```java
package co.edureka;

public interface State {
 public void changeState(Context context);

 }
```

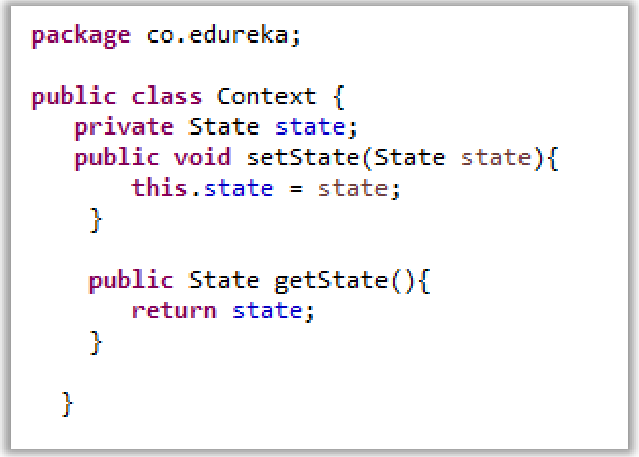


```java
package co.edureka;

public class InitialState implements State{

 public void changeState(Context context){
     context.setState(this);
   }

 }
```



```java
package co.edureka;

public class FinalState implements State {

 public void changeState(Context context){
     context.setState(this);
     }
  }
```

State Pattern Example(contd.)



```
package co.edureka;

public class Context {
   private State state;
   public void setState(State state){
       this.state = state;
    }

    public State getState(){
       return state;
    }

  }
```

In the previous slide we have one State interface, two classes InitialState and FinalState that implements State interface.

State interface have one method that takes a Context type object.

Here state of Context is changed using InitialState and FinalState changeState() method.

J2EE Patterns

DAO → It is a pattern which provides the API to interact with the database. DAO will have an interface. It uses the bean class to insert and retrieve data from the db.

DTO → This pattern is used when we need to send multiple attributes from client to server.

MVC → MVC Stands for Model View Controller. Model handles the data, view is for display purpose and controller is used to control the application.

Intercepting Filters → It is used when we want to do pre-processing and post-processing before and after a request processed. It is used for login authentication, logging etc.

Front Controller → This design pattern will have a centralized request handling mechanism so that all the requests will be handled by a single handler. This is like Struts servlet which is handled in web.xml.

Business Delegate → This pattern separates presentation tier and business logic tier.

J2EE Patterns (Contd.)

Context Object → Stores the information about incoming HTTP request like when a text document is sent who has sent it, when it is sent etc. It stores this information in map.

Application Controller → It is like Struts actions. That is the centralized controller of the application.

View Helper → When dynamic data to be displayed from model is required then view helpers are used. Jsp:usebean can be used for displaying data of the model.

Composite view → Display of view from multiple sub views like header, footer and table in the main layout.

Dispatcher view → This is a combination of view helper and composite view. View helper handles the client request and prepare the dynamic presentation as the response.

Service to worker → When a client request is handled by business logic before passing the request to the view. Struts actions. Request goes to action class and then to the view class.

J2EE Patterns (Contd.)

Service locator → Every time looking up using JNDI lookUP() is expensive. For the first time this pattern does the lookup and caches it. Next time it gives the cached object rather than lookup which improves the performance.

Session facade → Implemented as Session beans in EJB. It hides the implementation of Entity Beans. It takes care of client request and gets what the client needs.

Application Service → This pattern centralizes and aggregates the business patterns. An application service can be thought of helper to a session facade that take care of business logic and workflow.

Business Objects → This pattern separates business data with business logic. Can be considered as bean class and business logic class.

Composite entity → This pattern is used in EJB Persistence mechanism. Composite entity is a EJB entity bean which represents the graph of objects. When a composite entity is updated then its dependent bean objects also gets updated automatically as it is managed by Entiry bean of EJB.

J2EE Patterns (Contd.)

Transfer Object → This pattern uses an object to carry the data across the tiers.

T O Assembler → Transfer Object Assembler combines multiple transfer objects from various business components and services and return to its clients.

Value List Handler → This pattern caches the results and allows the client to search and traverse the items from this list. This pattern interacts directly with DAO and stores as Transfer Objects.

Service Activator → This pattern will receive asynchronous requests and invoke the required business methods and returns the data to the client.

Domain Store → User does not want container manager persistence or bean managed persistence. User wants to develop their own persistence model which is domain store design pattern.

Web Service Broker → Exposes many services using XML and web protocols. Web services are the example of this which will be discussed in the next module.

MVC Pattern Example

MVC(Model View Controller) pattern is one of the most popular pattern and has been their from a long time.

MVC separates the concerns like, code to present the view to user should be taken care by one component.
To write your business logic use some other component, and to frame your classes use another component.

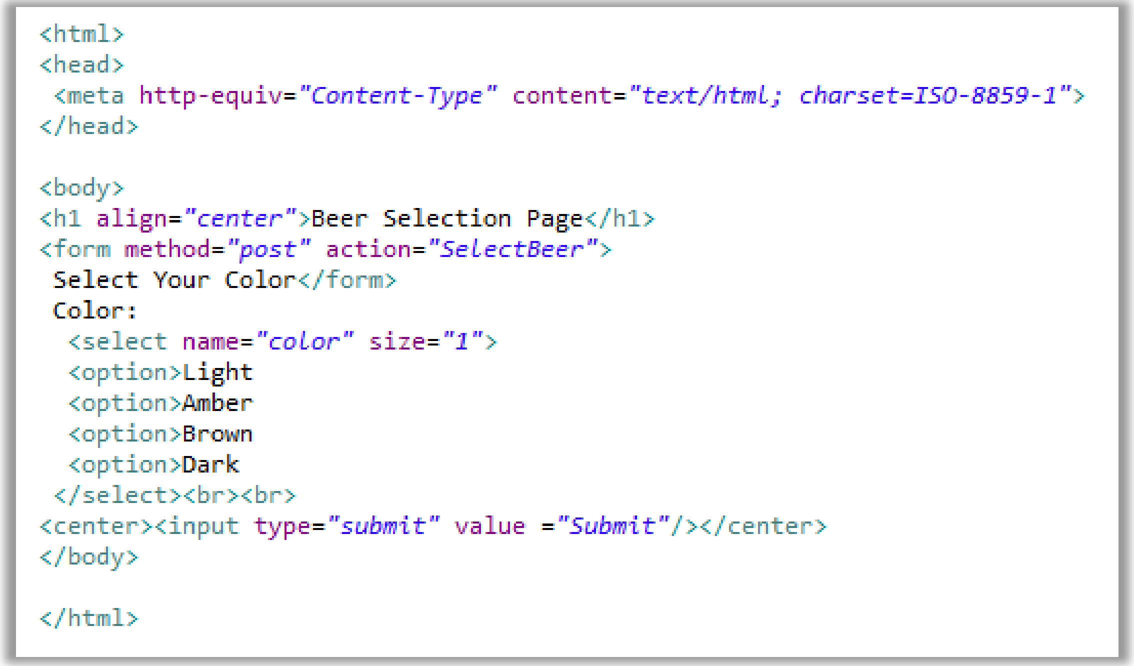
In J2EE JSP, Servlet and Java classes take the place of these three components.

→ View – JSP (Java Server Pages)

→ Model – Java Class

→ Controller – Servlet

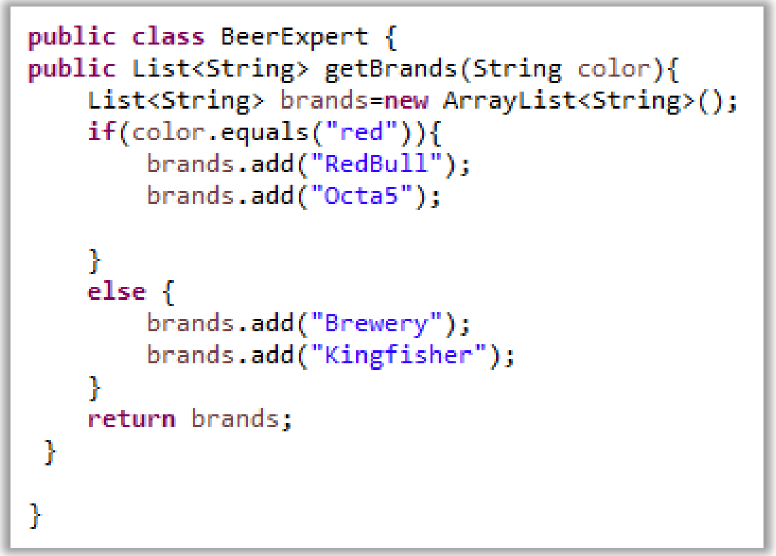
MVC Pattern Example (Contd.)



```
<html>
<head>
 <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
</head>

<body>
<h1 align="center">Beer Selection Page</h1>
<form method="post" action="SelectBeer">
 Select Your Color</form>
 Color:
  <select name="color" size="1">
  <option>Light
  <option>Amber
  <option>Brown
  <option>Dark
 </select><br><br>
<center><input type="submit" value ="Submit"/></center>
</body>

</html>
```

- This is our JSP page that works as view for the user.

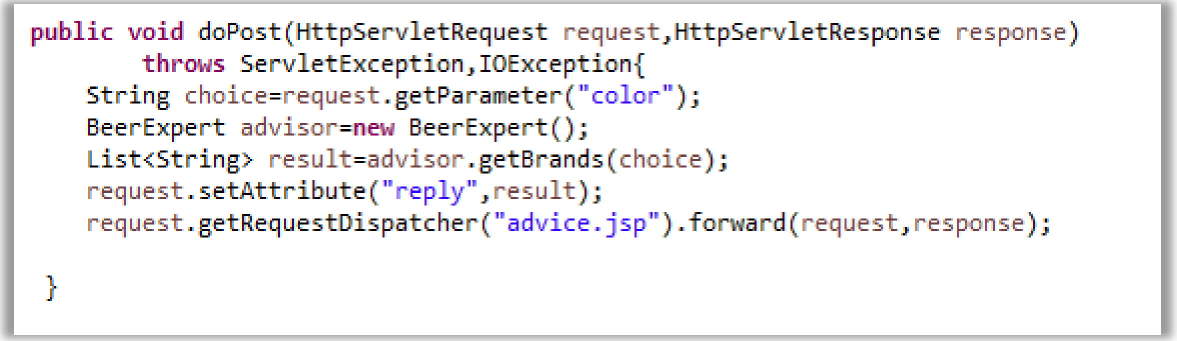
MVC Pattern Example (Contd.)



```
public class BeerExpert {
public List<String> getBrands(String color){
    List<String> brands=new ArrayList<String>();
    if(color.equals("red")){
        brands.add("RedBull");
        brands.add("Octa5");

    }
    else {
        brands.add("Brewery");
        brands.add("Kingfisher");
    }
    return brands;
 }

}
```

This is our Model class which have a method that takes a String argument and returns a List of String as an advice.

MVC Pattern Example (Contd.)



```
public void doPost(HttpServletRequest request,HttpServletResponse response)
        throws ServletException,IOException{
    String choice=request.getParameter("color");
    BeerExpert advisor=new BeerExpert();
    List<String> result=advisor.getBrands(choice);
    request.setAttribute("reply",result);
    request.getRequestDispatcher("advice.jsp").forward(request,response);

 }
```

This is our Servlet class that works as Controller in the application.

So when user selects his/her choice and clicks on submit button.

The control will go to this Servlet, which will call the required Model class.
Model class will return the result and controller will forward the control to some other view.

Annie’s Question

Please look at the program and can you say what will be output of program. Here base class variable is having the object of the derived class. There is display() function in both the classes. Which display method would be called?

Annie’s Answer

Derived class display() method would be called as the object is of derived type. Many times, at run time you will get to know the object type and the corresponding method of the class would be called, hence runtime polymorphism.

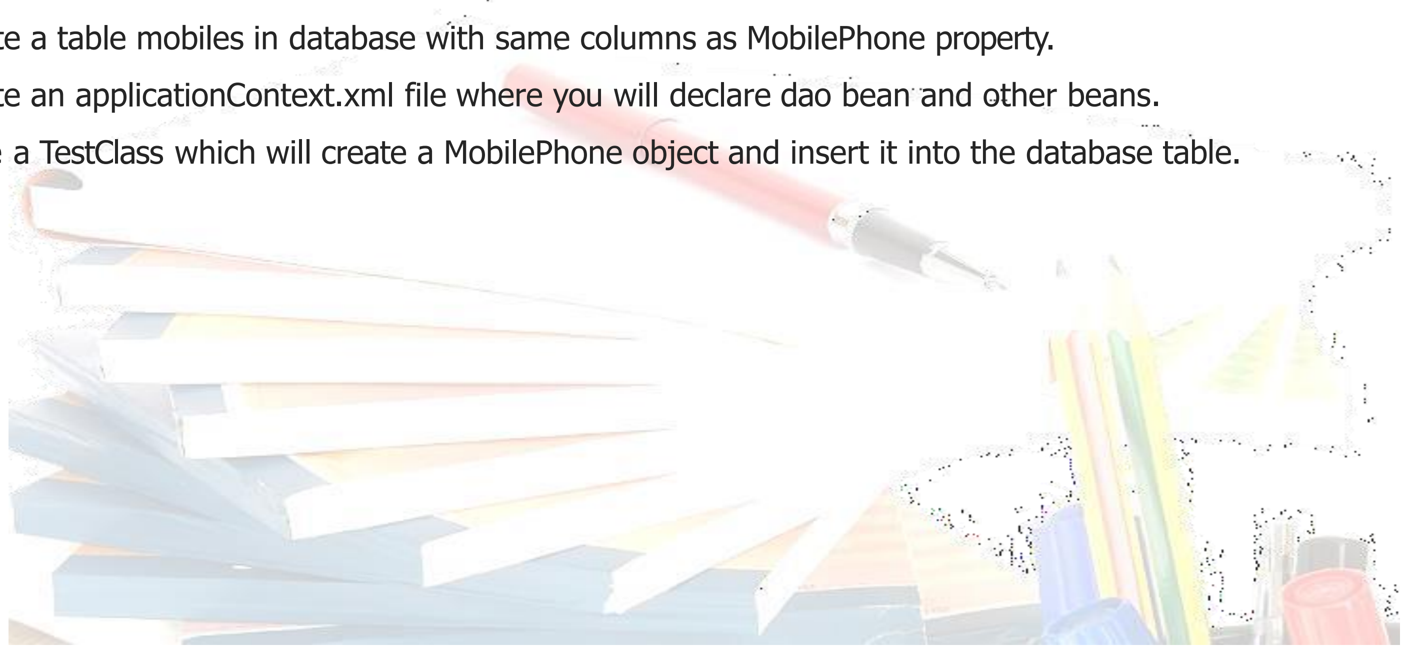
LAB

QUESTIONS

Assignment

Write a spring application, which has MobilePhone class and properties like manufacturer name, price, and colour.

1. Provide the setter and getter methods also in the same class.
2. Create a MobilePhoneDAO class which will have saveMobile, deleteMobile, updateMobile methods.
3. Create a table mobiles in database with same columns as MobilePhone property.
4. Create an applicationContext.xml file where you will declare dao bean and other beans.
5. Write a TestClass which will create a MobilePhone object and insert it into the database table.

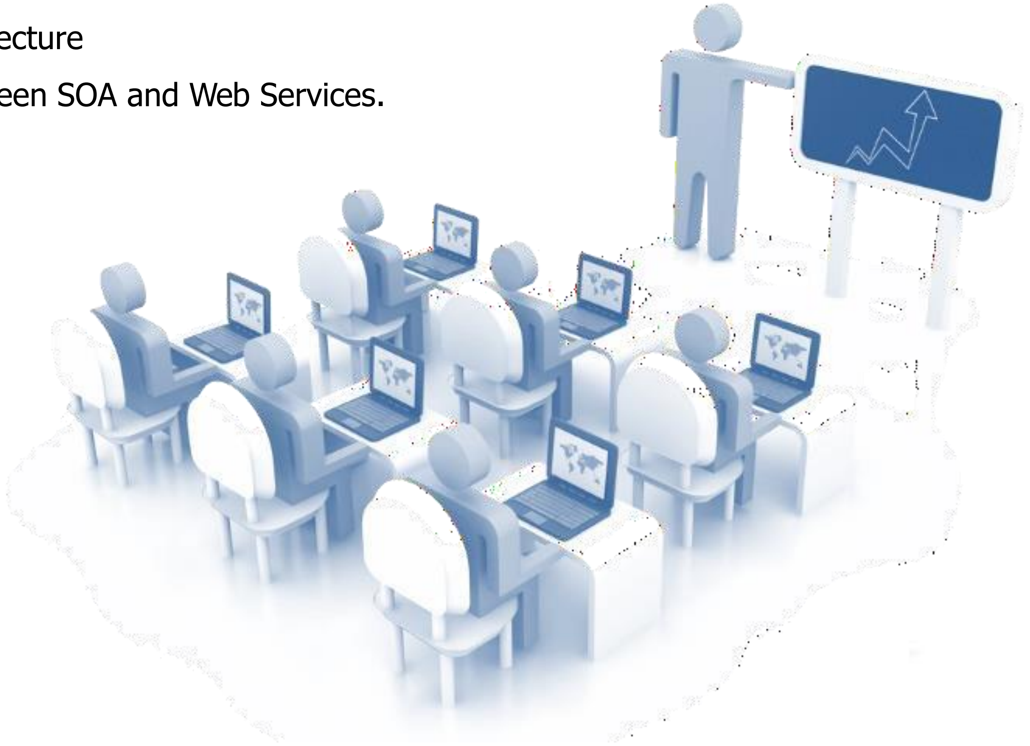


Agenda of the Next Module

In the next module we will be able to:

→ Understand SOA, its advantages and architecture

→ Learn how to use SOA and difference between SOA and Web Services.



Pre-work

- Preparation for the next module: Go through is web service.

Thank you!