SPRING

1. **Spring Basics**
    1. Why Spring
    2. What is Spring
    3. Inversion of Control
    4. Wiring Beans
    5. Aspect Oriented Programming
    6. Hitting the Database
2. **Spring Intermediate**
    1. Managing Transactions.
    2. Remoting.
    3. Accessing Enterprise Services.
    4. Building Web Layer.
3. **Spring Advanced**
    1. Web Framewok.
    2. Spring Validator Framework.
    3. Working with other frameworks.
    4. Securing Spring applications.

Why Spring ?

1. Drawbacks of J2EE Development.
2. Ejb is a Standard - Adv
    1. Wide industry Support
    2. Wide adoption
    3. Toolability
3. Complexities of EJB
    1. Writing an EJB is overly complicated
    2. Entity Ejb's fall Short
    3. Spring and EJB Common ground – next slide
4. Good Design is more important than the technology.
5. JavaBeans loosely coupled through interfaces is a good model.
6. Code should be easy to test

Why Spring ?

Spring and EJB common Ground

Feature	EJB	Spring
Transaction Management	Must use a JTA transaction Manager Supports transactions that span remote method calls	Supports multiple transaction environment through its PlatformTransactionManager Interface, including JTA, hibernate, JDO and JDBC Distributed Transactions – must be used with a JTA transaction manager
Declarative transaction support	Can be done through DD Use of * character Cannot declaratively define rollback behavior – this must be done programatically.	Can be done through spring config files or through meta data Use of regular expressions. Can declaratively define rollback behavior per method and per exception type
Persistance	Supports BMP and CMP	Provides a framework for integrating with several tech's like Hibernate, JDO etc

Why Spring ?

Spring and EJB common Ground

Feature	EJB	Spring
Declarative Security	Supports declarative security through users and roles. The management and implementation of users and roles is container specific. Declarative security is configured in the deployment descriptor	No Security implementation out-of-the box. Acegi, an open source security framework built on top of spring, provides declarative security through the spring configuration file or class metadata.
Distributed Computing	Provides Container-managed remote method calls.	Provides proxying for remote calls via RMI, JAX-RPC, and Web services.

What is Spring ?

Spring is a open-source framework, created by Rod Johnson.

It is designed to address the complexity of enterprise application development

Spring makes it possible to use plain-vanilla JavaBeans to achieve things that were previously possible with EJB's.

"Spring is a lightweight inversion of control and aspect-oriented container framework"

LightWeight ?

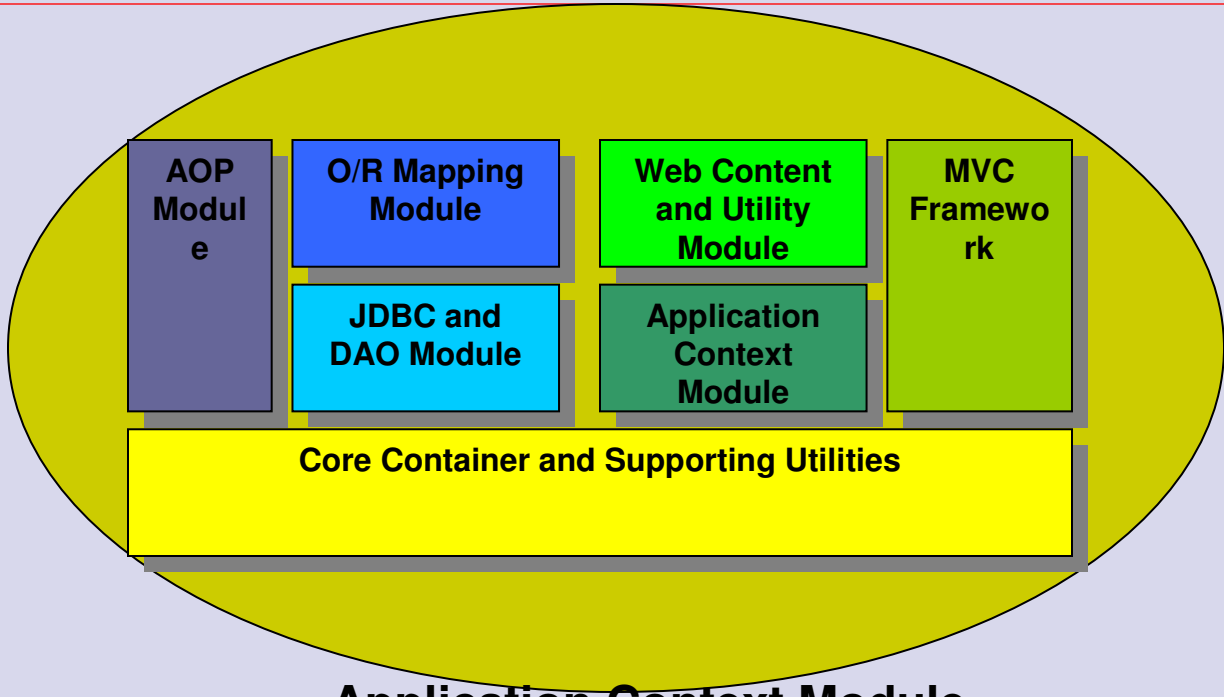
Inversion of Control ? – ex: ShoeFactory (for code Ref: L1)

Aspect Oriented ? – ex: Observer(for code Ref L2)

Container ? – ex: Environment (for code Ref L1)

Framework ?

What is Spring – Spring Modules ?



The Core Container	Application Context Module
Bean Factory the heart of spring based applications	This module extends the support of BeanFactory and makes spring a framework.
Implementation of Factory Pattern that applies to IOC.	Add Support for I18N, application life cycle events and validation
	Supplies many enterprise services such as JNDI access, EJB integration, remoting and scheduling.

How to Start -

Jar Files Needed


Asm.jar


commons-collections-2.1.1.jar


dom4j-1.6.1.jar


commons-logging.jar


spring-aspects.jar


cglib-2.1.3.jar


Spring.jar


spring-mock.jar

Create a Java Project and add the above minimum jar files in the class path,
If any more jar file is needed then pls add them from the JAR folder of the course CD
Add the below lines in a file called config.xml and place it in your project folder.

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

<beans>  </beans>
```

First Spring Application - IOC

1. Modify the ShoeFactory example to work for Spring.
2. As per the previous slide, create a project and add the necessary jar files In the class path and also add the config.xml in the project folder.
3. Make the following addition in the config.xml file

```
<bean id="fac" class="shoepack.LakhaniShoeFactory"></bean>
<bean id="shop" class="shoepack.RamuShoeShop">
    <property name="factory">
        <ref bean="fac"/>
    </property>
</bean>
```

4. Now Modify the scenario.java

```
public static void main(String[] args)throws Exception {
ApplicationContext ctx=
        new FileSystemXmlApplicationContext("config.xml");

ShoeShop shop=(ShoeShop)ctx.getBean("shop");
System.out.println(shop.getFactory());
System.out.println(shop.sellShoe());
}
```

Wiring Beans

1. BeanFactory

```
BeanFactory factory=
        new XmlBeanFactory(new FileInputStream("beans.xml"));
```

2. ApplicationContext

```
ApplicationContext context=
        new classPathXmlApplicationContext("myspring.xml");
                or
        new FileSystemXmlApplicationContext("myspring.xml");
                or
        new XmlWebApplicationContext("myspring.xml");
```

Bean Factory the heart of spring based applications	This module extends the support of BeanFactory and makes spring a framework.
Implementation of Factory Pattern that applies to IOC.	Add Support for I18N, application life cycle events and validation
	Supplies many enterprise services such as JNDI access, EJB integration, remoting and scheduling.

Beans Life

Instantiate - populate properties - BeanNameAwares setBeanName() - BeanFactoryAwares setBeanFactory() - Pre Initialization - Initializing Beans - Call Custom init method - Post Initialization (Bean is ready to use) - DisposableBean's destroy() - Call destroy()

ApplicationContext Life

Instantiate - populate properties - BeanNameAwares setBeanName() - BeanFactoryAwares setBeanFactory() - ApplicationContextAware setApplicationContext() - Pre Initialization - Initializing Beans - Call Custom init method - Post Initialization (Bean is ready to use) - DisposableBean's destroy() - Call destroy()

Wiring Beans

1. **Prototyping Vs Singleton – default is singleton**

```
<bean id="bean2" class="shoepack.LakhaniShoeFactory"
        singleton="false"/>
```

2. Initialization and destruction

```
<bean id="bean2" class="shoepack.LakhaniShoeFactory"
        init-method="setup" destroy-method="teardown"/>
```

3. Injecting dependencies via setter methods

```
<beans>
<bean id="bean1" class="shoepack.ShoeShop">
<property name="name"><value>Haaris</value></property></bean>
</beans>
```

4.Referencing Other Beans

```
<beans>
<bean id="bean1" class="shoepack.ShoeShop">
        <property name="factory"><ref local="bean2"/></property>
</bean>
<bean id="bean2" class="shoepack.LakhaniShoeFactory"></bean>
</beans>
```

Wiring Beans

Wiring Collections

Collections supported by Spring's wiring

- <list> - java.util.List, arrays
- <set> - java.util.Set
- <map> - java.util.Map
- <props> - java.util.Properties

Wiring lists and arrays

```
<property name="nameList">
        <list>
                <value>bar1</value>
                <ref bean="bar2"/>
        </list>
</property>
```

Wiring sets

```
<property name="nameList">
        <set>
                <value>bar1</value>
                <ref bean="bar2"/>
        </set>
</property>
```

Wiring Beans

Wiring maps

```
<property name="nameList">
    <map>
        <entry key="key1">
            <value>bar1</value>
        </entry>
        <entry key="key2">
            <ref bean="bar2"/>
        </entry>
    </map>
</property>
```

Wiring properties

```
<property name="nameList">
    <props>
<prop key="key1">bar1</prop>
<prop key="key2"><null/></prop> -- setting null values
    </props>
</property>
```

Wiring Beans

Injecting dependancies via Constructor

```
<beans>
        <bean id="bean1" class="shoepack.ShoeShop">
            <constructor-arg>
<value>44</value> or <ref bean="bar"/>
            </constructor-arg>
        </bean>
</beans>
```

Handling ambiguous constructor arguments

```
<beans>
  <bean id="bean1"
class="shoepack.ShoeShop">
    <constructor-arg index="1">
      <value>44</value>
    </constructor-arg>
    <constructor-arg index="0">
      <value>44</value>
    </constructor-arg>
  </bean>
</beans>
```

OR

```
<beans>
  <bean id="bean1" class="shoepack.ShoeShop">
        <constructor-arg type="java.lang.String>
            <value>44</value>
        </constructor-arg>
        <constructor-arg type="java.lang.Integer">
            <value>44</value>
        </constructor-arg>
  </bean>
</beans>
```

Wiring Beans

Autowiring

```
<beans>
        <bean id="bean1" class="shoepack.ShoeShop"
autowire="byName"/>
</beans>
or
<beans>
        <bean id="bean1" class="shoepack.ShoeShop"
autowire="constructor"/>
</beans>
or
<beans>
        <bean id="bean1" class="shoepack.ShoeShop"
autowire="autodetect"/>
</beans>
```

By setting autowire to autodetect, you instruct the Spring container to attempt to autowire by constructor first. If it can't find a suitable match between constructor arguments and beans, it will then try to autowire using byType.

Wiring Beans

Mixing Auto and explicit wiring

```
<beans>
    <bean id="bean1" class="shoepack.ShoeShop"
autowire="byName">
        <property name="courseDAO">
            <ref bean="some bean"/>
        </property>
    </bean>
</beans>
```

Autowiring by default

```
<beans default-autowire="byName" >
```

Wiring Beans-PostProcessingBean

Working with Spring's Special Beans

The postProcessingBeforeInitialization() method is called immediately prior to bean initialization (the call to afterPropertiesSet() and the bean's custom init-method). Likewise, the postProcessAfterInitialization() method is called immediately after initialization.

```
public class MyPostProcessor implements BeanPostProcessor
{
public Object postProcessAfterInitialization(Object bean,String name)throws
BeansException
{
Field[] fields=bean.getClass().getDeclaredFields();
        try{
for(int i=0;i<fields.length;i++)        {
        if(fields[].getType().equals(java.lang.String.class))  {
        fields[i].setAccessible(true);
        String original=(String)fields[i].get(bean);
        fields[i].set(bean,fuddify(original));  }
        }}catch(Exception e){}   return bean;                    }

public String fuddify(String orig) {
if(orig == null) return orig; return orig.replaceAll("(r|l)","w").replaceAll(" (R|L)","w"); }
public Object postProcessBeforeInitialization(Object bean,String name)throws Exception    {
                return bean;                    }
}
```

Wiring Beans-PostProcessingBean

Registering bean post processors

If your application is running within a bean factory, you'll need to programmatically register each BeanPostProcessor using the factory's addBeanPostProcessor() method.

```
BeanPostProcessor fuddifier=new Fuddifier();
factory.addBeanPostProcessor(fuddifier);
```

If you're using an application context, you'll only need to register the post processor as a bean within the context.

```
<bean id="fuddifier" class="com.Fuddifier"/>
```

The container will recognize the fuddifier bean as a BeanPostProcessor and call its postprocessing methods before and after each bean is initialized.

As a result of the fuddifier bean, all string properties of all beans will be fuddified. For example, you had the following bean defined in XML.

```
<bean id="bugs" class="aaaa">
        <property name="des">
                <value>That is really a rabbit</value>
        </property>
</bean>
```

When the fuddifier processor is finished, the description property will hold "The weawwy wabbit.

Wiring Beans-PropertyPlaceHolder

The properties file

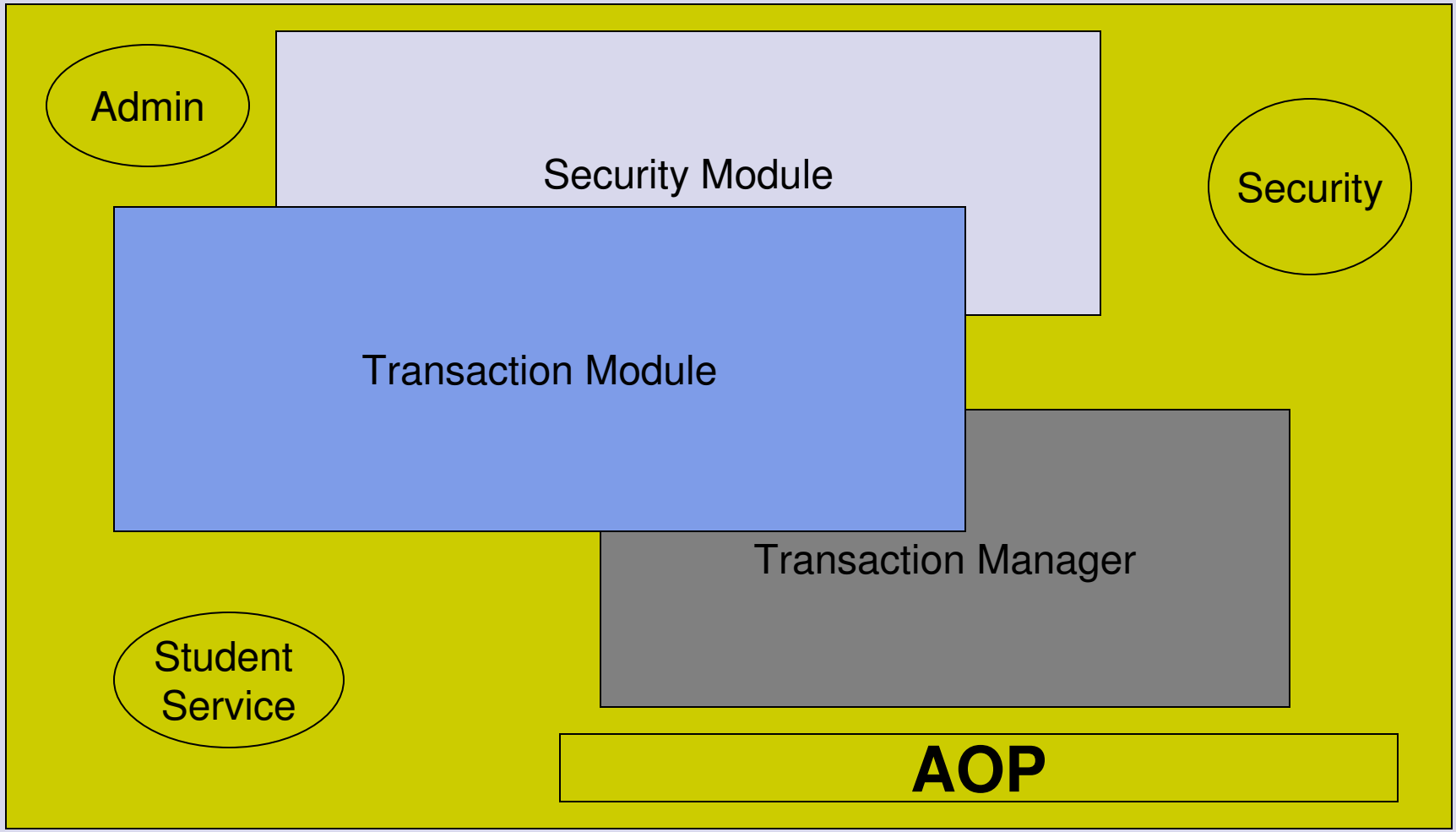
database.url=jdbc:hsqlib:Training
database.driver=org.hsqldb.jdbcDriver

To enable reading the above properties file, configure the following bean in your bean wiring file.

```
<bean id="propertyConfigurer"
class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer
">
        <property name="locations">
                <list>
                        <value>jdbc.properties</value>
                        <value>security.properties</value>
                </list>
        </property>
        or
        <property name="location">
                        <value>jdbc.properties</value>
                </property>
        </bean>
```

Aspect-Oriented Programming

While Inversion of control makes it possible to tie software components together loosely, Aspect Oriented programming enables you to capture functionality that is used throughout your application in reusable components.



Aspect-Oriented Programming

Creating Aspects

1. Creating Advice

a. Around - org.aopalliance.intercept.MethodInterceptor - Intercepts calls to the target method

b. Before - org.springframework.aop.MethodBeforeAdvice - Called before the target method is invoked

c. After - org.springframework.aop.AfterReturningAdvice - Called after the target method returns

d. Throws - org.springframework.aop.ThrowsAdvice - Called when a method throws an exception

Refer Lab2 for implementations and examples on the above.

Aspect-Oriented Programming

```
public class WelcomeAdvice implements
MethodBeforeAdvice {
                public void before(Method method,Object[]
args, Object target)
                {
                        Customer customer=(Customer)args[0];
- Calls first argument to Customer
                        System.out.println("Welcome Mr.
:"+customer.getName());
                }
        }
```

Aspect-Oriented Programming

```
<beans>
        <bean id="ShoeShopTarget" class="com.RamuShoeShop"/>
        <bean id="welcomeAdvice" class="com.WelcomeAdvice"/>
        <bean id="ShoeShop"
class="org.springframework.aop.framework.ProxyFactoryBean">
                <property name="proxyInterfaces">
                        <value>com.ShoeShop</value>
                </property>
                <property name="interceptorNames">
                        <value>welcomeAdvice</value>
                </property>
                <property>
                        <list>
                                <value>welcomeAdvice</value>
                        </list>
                </property>
                <property name="target">
                        <ref bean="ShoeShopTarget"/>
                </property>
        </bean>
</beans>
```

Aspect-Oriented Programming

```
public class ThankYouAdvice implements AfterReturningAdvice {
        public void afterReturning(Object returnValue, Method
method, Object[] args, Object target)
        {                       }
}

public class MyInterceptor implements MethodInterceptor
{
        private Set customers=new HashSet();
public Object invoke(MethodInvocation invocation)throws Throwable
        {
Customer customer=(Customer)invocation.getArguments()[0];
if(customer.contains(customer)
        {
        throw new ShoeException("one per customer");
        }
                Object shoe =invocation.proceed();
                customers.add(customer);
                return shoe;
        }
}
```

Aspect-Oriented Programming

```
public class ShoeExceptionAdvice implements ThrowsAdvice
{
        public void afterThrowing(ShoeException se){    ------logic ------}
        public void afterThrowing(Method m,Object[] args,Object
target,Throwable thro){}
}
```

Defining Pointcuts

Pointcuts determine if a particular method on a particular class matches a particular criterion. If the method is indeed a match then advice will be applied to this method. Spring's pointcuts allow us to define where our advice is woven into our classes in a very flexible manner.

Aspect-Oriented Programming

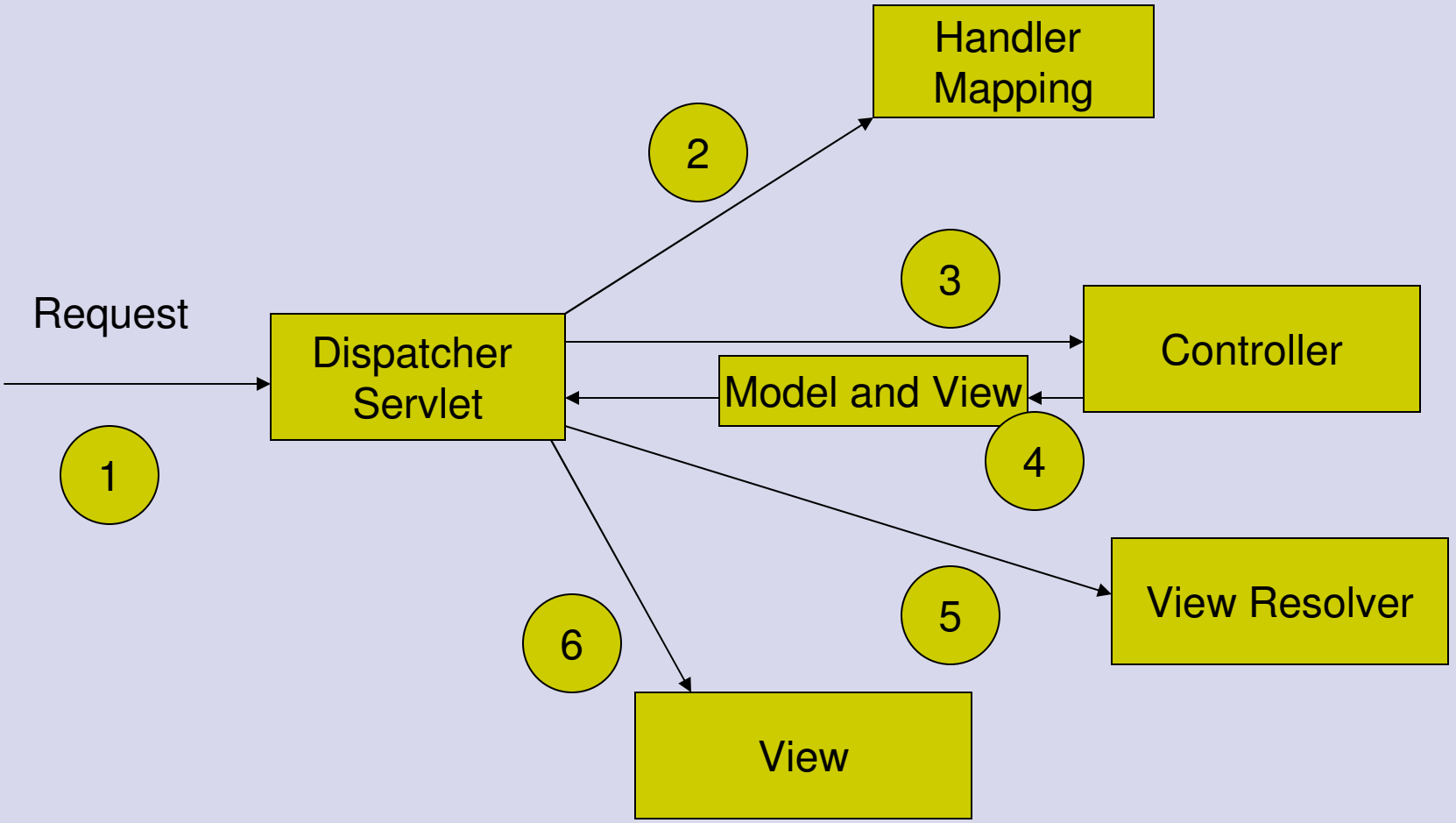
Defining a pointcut

```
        <bean id="ShoeShopTarget" class="com.RamuShoeShop"/>
        <bean id="welcomeAdvice" class="com.WelcomeAdvice"/>
<bean id=”pointcutadvisor”
class=”org.springframework.aop.support.NameMatchMethodPointcutAdvisor”
>
        <property name=”mappedName”><value>order*</value></property>
        <property name=”advice”><ref bean=”welcomeAdvice”/></property>
</bean>
<bean id="ShoeShop“
class="org.springframework.aop.framework.ProxyFactoryBean">
<property name="proxyInterfaces"><value>com.ShoeShop</value></property>
<property
name="interceptorNames"><value>pointcutadvisor</value></property>
<property name="target"><ref bean="ShoeShopTarget"/></property>
</bean>
</beans>
```

Instead of supplying wildcard characters, we just as easily explicity name each of the methods.

```
        <property name=”mappedNames”>
                <list>  <value>order</value><value>order2</value></list>
        </property>
```

Spring MVC


Handler Mapping
2
3
Request
Dispatcher Servlet
Controller
Model and View
1
4
View Resolver
5
6
View

Spring MVC

Steps to Configure

1. Make the following entries in your web.xml

```
<context-param>
<param-name>contextConfigLocation</param-name>
<param-value>/WEB-INF/training-service.xml,
/WEB-INF/training-data.xml</param-value>
</context-param>

<!-- listener>
<listener-class>org.springframework.web.context.ContextLoaderListener
</listener-class>
</listener -->
<!-- listener>
<listener-class>org.springframework.web.util.Log4jConfigListener
</listener-class>
</listener -->
```

Spring MVC

Steps to Configure
Contd..

```
<!-- JSPC servlet mappings start -->
<servlet>
<servlet-name>context</servlet-name>
<servlet-class>org.springframework.web.context.ContextLoaderServlet
</servlet-class>
<load-on-startup>1</load-on-startup>
</servlet>
<servlet>
        <servlet-name>training</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet
       </servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>training</servlet-name>
        <url-pattern>*.htm</url-pattern>
    </servlet-mapping>
```

Spring MVC

First Example

1. Write the controller class that performs the logic behind the homepage.
2. Configure the controller in the DispatcherServlet's context configuration file (training-servlet.xml)
3. Configure a view resolver to tie the controller to the jsp.
4. Write the jsp that will render the homepage to the user.
5. Make sure you have the following jar files in the classpath of web app.


Asm.jar


cglib-2.1.3.jar


commons-logging.jar


ommons-collections-2.1.1.ja


dom4j-1.6.1.jar


Spring.jar


spring-aspects.jar


spring-mock.jar

Spring MVC

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

<beans>
<bean name="/home.htm"
class="controlpack.HomeController">
<property name="greeting">
<value>Welcome to Spring Training!!!!!!!...!!!!!</value>
</property>
</bean>

<bean id="viewResolver"
class="org.springframework.web.servlet.view.InternalResourceViewResolver">
<property name="prefix">
<value>/views/</value>
</property>
<property name="suffix"> <  value>.jsp</value>  </property>
</bean></beans>
```

Haaris Infotech

Driven by Technology

Spring MVC

```
package controlpack;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.Controller;

public class HomeController implements Controller{

    private String greeting;
    public ModelAndView handleRequest
    (HttpServletRequest req, HttpServletResponse res) throws Exception {
        return new ModelAndView("home","message",greeting);
    }
    public void setGreeting(String greeting)
    {
        this.greeting=greeting;
    }
}
```

Spring MVC

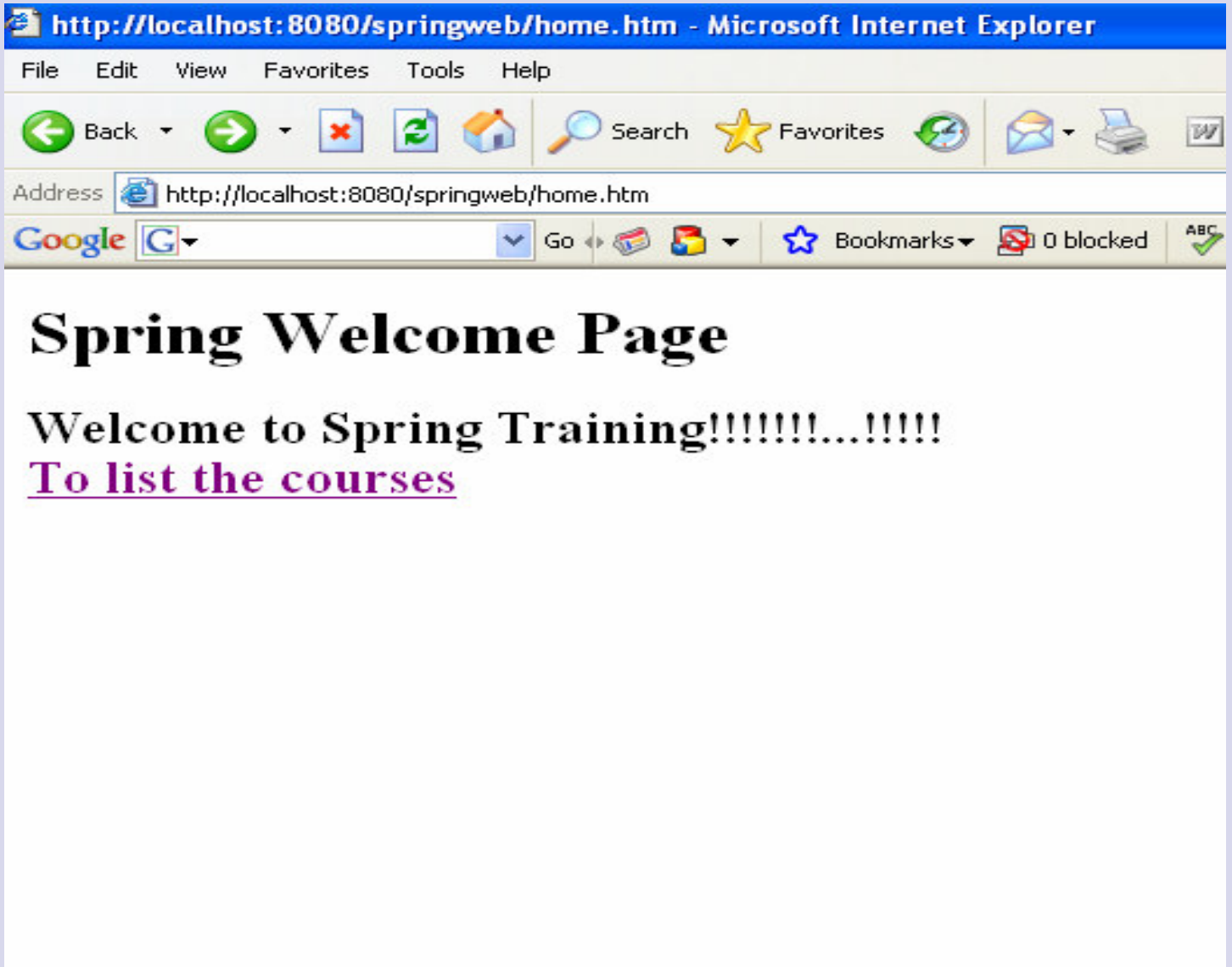
```
<h1>

Spring Welcome Page

</h1>

<h2><%=request.getAttribute("message")%>
<br>


<a href="/springweb/listcourses.htm">To list the courses </a>
```

Now Start tomcat and browser, change your url to
http://localhost:8080/springweb/home.htm

Haaris Infotech

Driven by Technology

Spring MVC


http://localhost:8080/springweb/home.htm - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://localhost:8080/springweb/home.htm
Google Go Bookmarks 0 blocked
Spring Welcome Page
Welcome to Spring Training!!!!!!!...!!!!!
To list the courses

Spring MVC

Mapping request to Controllers

1. BeanNameUrlHandlerMapping – the default

2. SimpleUrlHandlerMapping

3. Example

```
<bean id="simpleUrlMapping"
class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
<property name="mappings">
<props>
<prop key="/home.htm">HomeController</prop>

</props>
</property>
</bean>
```

Spring MVC

3. Using metadata to map controllers.

```
<bean id="urlMapping" class="org.springframework.web.servlet.handler.
    metadata.CommonsPathMapHandlerMapping"/>

/**
• @@org.springframework.web.servlet.handler.commonsattributes.
   PathMap("/home.htm")

*/
Public class HomeController extends Controller
{
....
}
```

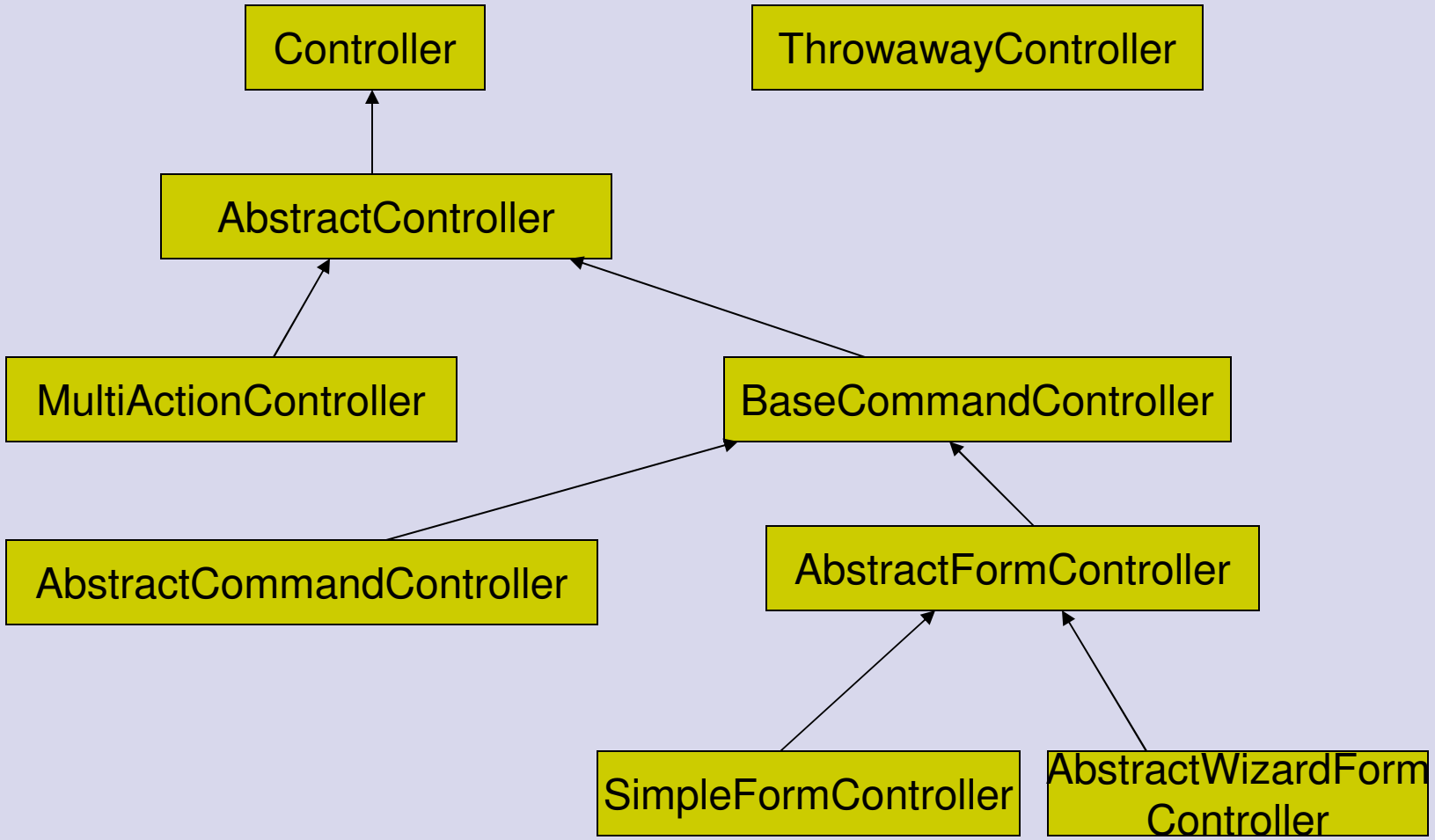
Spring MVC

Working with Multiple Handler mappings

```
<bean id=“beanNameUrlMapping” class=“org.springframework.web.servlet.
    handler.BeanNameUrlHandlerMapping”>
    <property name=“order”><value>1</value></property>
</bean>


<bean id="simpleUrlMapping"
class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
<property name=“order”><value>0</value></property>
<property name="mappings">
<props>
<prop key="/home.htm">HomeController</prop>

</props>
</property>
</bean>
```

Spring MVC

Handling requests with controllers.



Spring MVC

Simple	Controller (interface) AbstractController	Your controller is extremely simple, requiring little more functionality than is afforded by basic java servlets.
Throw away	ThrowawayController	You want a simple way to handle requests as commands
Multi-Action	MultiActionController	Your application has several actions that perform similar to related logic
Command	BaseCommandController AbstractCommandController	Your controller will accept one or more parameters from the request and bind them to an object. Also capable for performing parameter validation
Form	AbstractFormController SimpleFormController	You need to display an entry form to the user and also process the data entered into the form
Wizard	AbstractWizardFormController	Walk through a complex multipage entry form that ultimately get processed as a single form

Spring MVC

Example 2

```
package controlpack;
public class ListCoursesController extends AbstractController{
        private CourseService courseService;
        protected ModelAndView handleRequestInternal
(HttpServletRequest arg0, HttpServletResponse arg1) throws Exception {
                Set courses=courseService.getAllCourses();
                return new ModelAndView("courseList","courses",courses);
        }
        public void setCourseService(CourseService courseService)
        {
                this.courseService=courseService;
        }

}
```

Spring MVC

Example 2

```
<bean id="courseService" class="controlpack.CourseService"/>
<bean id="ListCourses" class="controlpack.ListCoursesController">
<property name="courseService">
<ref bean="courseService"/>
</property>
</bean>
```

Plus an entry into the simple mapping

```
<prop key="/listcourses.htm">ListCourses</prop>
```

Spring MVC

Example 2

```
public class CourseService {
        public Set getAllCourses()
        {
                TreeSet tset=new TreeSet();
        tset.add("JAVA"); tset.add("XML");tset.add("JEE");tset.add("SPRING");
        tset.add("AJAX");               tset.add("HIBERNATE");
                return tset;
        }
        public Course getCourse(Integer id)
        {
                switch(id.intValue())
                {
                case 1:{return new JavaCourse();}
                case 2:{return new XMLCourse();}
                default:{return new JavaCourse();}
                }
        }
}
```

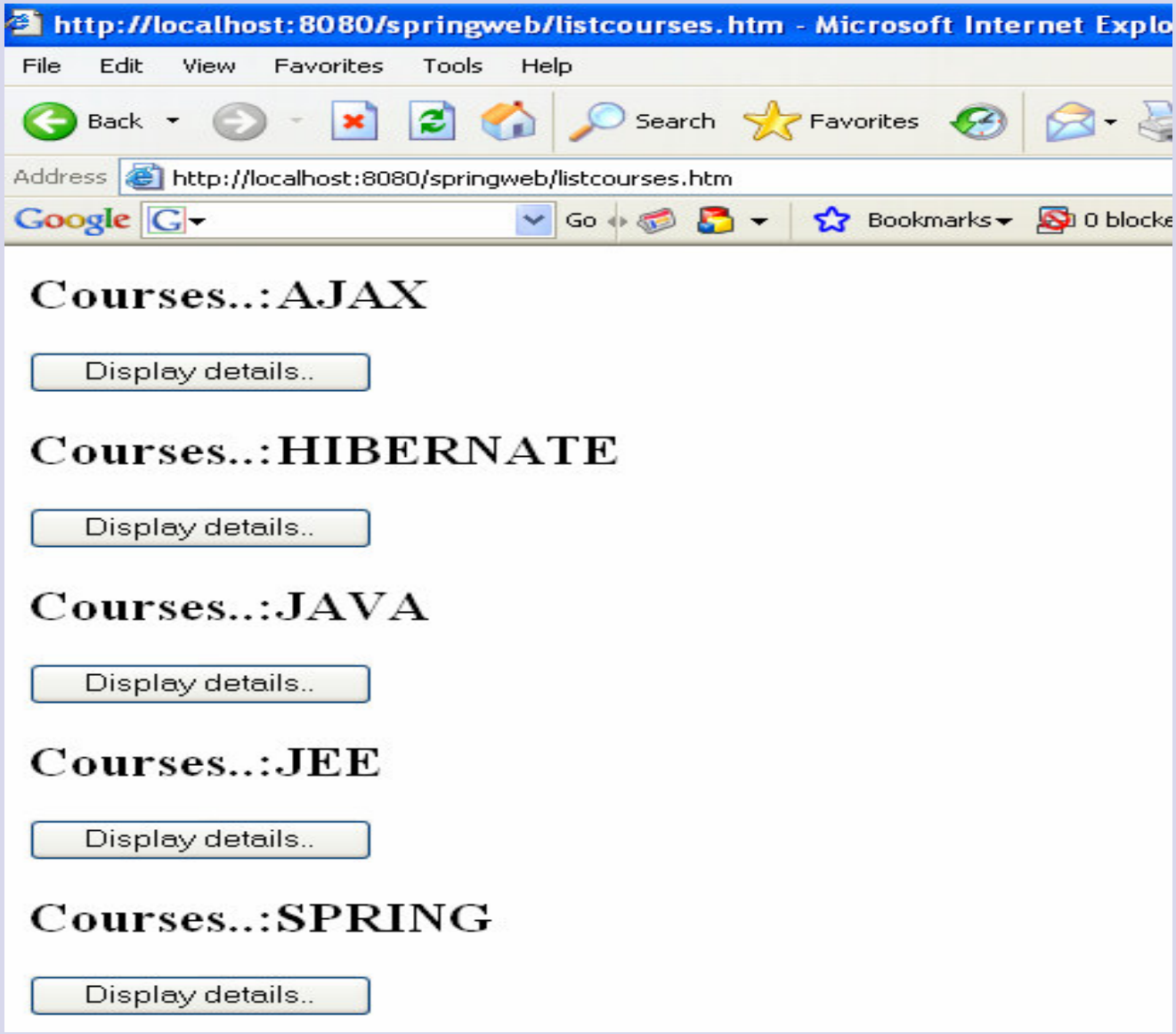
Spring MVC

Example 2

CourseList.jsp

```
<%@ page import="java.util.*"%>
<%
TreeSet s=(TreeSet)request.getAttribute("courses");
Iterator i=s.iterator();
int id=0;
while(i.hasNext())
{
out.println("<h2>Courses..:"+i.next().toString()+"<br>");
id++;
%>
<form action="/springweb/dispcourse.htm">
<input type="hidden" name="id" value="<%=id%>">
<input type="submit" value="Display details..">
</form>
<% }
%>
```

Spring MVC



Processing Commands

For instance after viewing the list of available courses, you may want to view more details about that course. The controller that display course information will need to take the Id of the course as parameter.

Spring MVC

```
public class DisplayCourseController extends AbstractCommandController{
        public DisplayCourseController()
        {
                setCommandClass(DisplayCourseCommand.class);
        }
        protected ModelAndView handle(HttpServletRequest req,
                        HttpServletResponse res, Object command,
BindException errors) throws Exception {
                DisplayCourseCommand
dispCommand=(DisplayCourseCommand)command;
                Course course=courseService.getCourse(dispCommand.getId());
                return new ModelAndView("courseDetail","course",course);
        }
private CourseService courseService;
        public CourseService getCourseService() {
                return courseService;
        }
        public void setCourseService(CourseService courseService) {
                this.courseService = courseService;
        }
}
```

Spring MVC

```
public class DisplayCourseCommand {
public DisplayCourseCommand(){}
private Integer id;
public Integer getId() {        return id;    }
public void setId(Integer id) {
            this.id = id;
}

}

public abstract class Course {
            public abstract String getDetails();
}

public class XMLCourse extends Course{

            public String getDetails() {
                        // TODO Auto-generated method stub
                        return "XML Course : - 3 days - Fees: 3000 Rs.";
}}
```

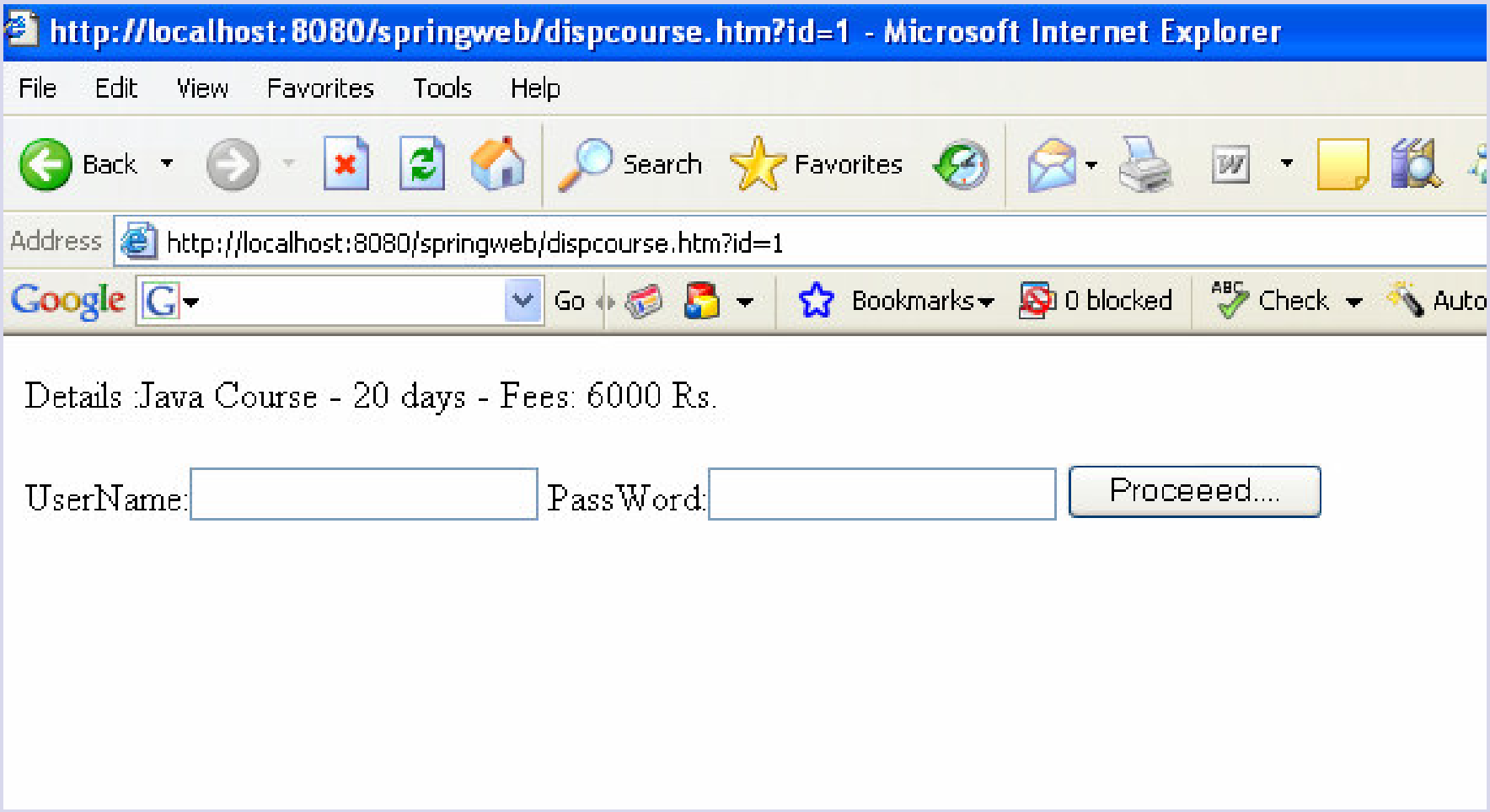
Spring MVC

```
<bean id="DispCourse" class="controlpack.DisplayCourseController">
<property name="courseService">
<ref bean="courseService"/>
</property>
</bean>
```

CourseDetail.jsp

```
<%@ page import="controlpack.*"%>
<%
Course obj=(Course)request.getAttribute("course");
out.println("Details :"+obj.getDetails());

%>
<form action="/springweb/register.htm">
UserName:<input type="text" name="uname">
PassWord:<input type="text" name="upass">
<input type="submit" value="Proceeed....">
</form>
```

Spring MVC


http://localhost:8080/springweb/dispcourse.htm?id=1 - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://localhost:8080/springweb/dispcourse.htm?id=1
Google Go Bookmarks 0 blocked Check Auto
Details :Java Course - 20 days - Fees: 6000 Rs.
UserName: PassWord: Proceeed....

Processing Form Submissions

Form controllers take the concept of command controllers a step further by adding functionality to display a form when an GET request is received and process the form when a POST is received.

```
public class RegisterUserController extends SimpleFormController{
        public RegisterUserController()
        {
                setCommandClass(Student.class);
        }
protected ModelAndView onSubmit(HttpServletRequest arg0,
HttpServletResponse arg1, Object arg2, BindException arg3) throws Exception {
        Student student=(Student)arg2;
        studentService.enrollStudent(student);
        return new ModelAndView(getSuccessView(),"student","student");
}
```

Spring MVC

```
package controlpack;

public class Student {
 String uname,upass;
public String getUname() {
        return uname;
}

public void setUname(String uname) {
        this.uname = uname;
}
public String getUpass() {
        return upass;
}
public void setUpass(String upass) {
        this.upass = upass;
}
}
```

Spring MVC

```
package controlpack;
public class StudentService {
        public StudentService()
        {
                System.out.println("Student Service object created....");
        }

        public void enrollStudent(Student s)
        {
                System.out.println("Student is enrolled.....");
        }
}
```

Spring MVC

```
<bean id="studentService" class="controlpack.StudentService"/>

<bean id="Register" class="controlpack.RegisterUserController">
<property name="studentService">
<ref bean="studentService"/>
</property>
<property name="formView">
<value>courseDetail</value>
</property>
<property name="successView">
<value>studentWelcome</value>
</property>
<!-- property name="validator">
<bean class="controlpack.StudentValidator"/>
</property -->
</bean>
```

<prop key="/register.htm">Register</prop> make this entry in simepleurlmapping

Also create studentWelcome.jsp

Spring MVC

How to validate the Form input

When the registercontroller calls enrollStudent(), it is important to ensure that all the data in the student command is valid and complete.

The org.springframework.validation.Validator interface accomodates validation for spring MVC

```
Public interface Validator {
        void validate(Object ob, Errors errors);
        boolean supports(Class clazz);

}
```

Implementation of this interface should examine the fields of the object passed into the validate() method and reject any invalid values via the Errors object.

The supports method is used to help spring determine whether or not the validator can be used for the given class.

Spring MVC

```
public class StudentValidator implements Validator{
        public boolean supports(Class claz) {
                return claz.equals(Student.class);
        }
        public void validate(Object command, Errors errors) {
                System.out.println("validate called.....");
                Student student=(Student)command;
        ValidationUtils.rejectIfEmpty(errors,"uname","required.uname","usernam
e is required");
        }
        private static final String password="^[a-zA-Z]{1}[a-zA-Z0-9_]*$";
        private void validateUpass(String upass,Errors errors)
        {
        ValidationUtils.rejectIfEmpty(errors,"upass","required.upass","password
is required..");
                Perl5Util p=new Perl5Util();
                if(!p.match(password,upass))
        {
                errors.reject("invalid.password","password is invalid")
        }       }       }
```

Spring MVC

The other thing to do is to use the StudentValidator with RegisterUserController.s
You can do this by wiring a student validator bean into the RegisterUserController bean.

```
<bean id="Register" class="controlpack.RegisterUserController">
<property name="studentService">
<ref bean="studentService"/>
…..
<property name="validator">
<bean class="controlpack.StudentValidator"/>
</property>
</bean>
```

Spring MVC

Building a basic wizard controller

In form processing, instead of creating one huge form, lets break the form into several subforms and walk through as we do in wizards.

To construct a Wizard controller

```
class FeedbackWizardController extends AbstractWizardFormController{
        public FeedbackWizardController()
        {
                setCommandClass(FormCommand.class);
        }
        protected ModelAndView processFinish(HttpServletRequest arg0,
HttpServletResponse arg1, Object arg2, BindException arg3) throws Exception {
                return new ModelAndView("thankyou");
        }
        protected ModelAndView processCancel(HttpServletRequest arg0,
HttpServletResponse arg1, Object arg2, BindException arg3) throws Exception {
                return new ModelAndView("home");
        }}
```

Spring MVC

Building a basic wizard controller

```
public class FormCommand {
String name1,name2;

public String getName1() {
        return name1;
}

public void setName1(String name1) {
        this.name1 = name1;
}

public String getName2() {
        return name2;
}

public void setName2(String name2) {
        this.name2 = name2;
}}
```

Spring MVC

```
<bean id="feedbackController" class="controlpack.FeedbackWizardController">
<property name="pages">
<list>
<value>general</value>
<value>instructor</value>
</list>
</property>
</bean>
```

The first page to be shown in any wizard controller will be the first page in the list given to the pages property.

Spring MVC

To determine which page to go to the next, the controller consults its getTargetPage() method. This method returns an int, which is an index into the zero based list of pages given to the pages property.

The getTargetPage() method reads the request parameter name whose name begins with "_target" and ends with a number, the method removes the prefix and takes the number and take you to the page corresponding to the index.

To create a Next and Back button on the page, all you must do is create submit buttons that are appropriately named with the "_target" prefix.

General.jsp

```
<form method="post" action="/springweb/feedback.htm">
<input type="submit" value="next" name="_target1">
</form>
```

To Finish or cancel the wizard

Instructor.jsp

```
<form method="post" action="/springweb/feedback.htm">
<input type="submit" value="previous" name="_target0">
<input type="submit" value="finish" name="_finish">
<input type="submit" value="cancel" name="_cancel">
</form>
```

You should have the processCancel and processFinish methods in your controller.

Spring MVC

Handling multiple action in one controller

Steps to do.

```
public class MultiAction extends MultiActionController{
        public MultiAction(){          }
        public ModelAndView firstAction(HttpServletRequest
req,HttpServletResponse res)
        {
                System.out.println("First action called...");
                return new ModelAndView("thankyou");
        }
        public ModelAndView secondAction(HttpServletRequest
req,HttpServletResponse res)
        {
                System.out.println("second action called...");
                return new ModelAndView("thankyou");
        }
}
```

Spring MVC

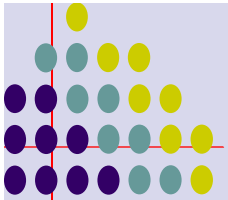
How to resolve method names

```
<bean id="multiaction" class="controlpack.MultiAction">
<property name="methodNameResolver">

<ref bean="methodNameResolver"/>
</property>
</bean>
<bean id="methodNameResolver"
class="org.springframework.web.servlet.mvc.multiaction.ParameterMethodName
Resolver">
<property name="paramName">
<value>action</value>
</property>
</bean>

<bean id="simpleUrlMapping"
…..
<prop key="/multiaction.htm">multiaction</prop>
….
</bean>
```

The paramName property indicates the name of the request parameter that will contain the name of the execution method to choose. In this case, it has been set to action.

http://localhost:8080/springweb/multiaction.htm?action=secondAction

Spring MVC



Haaris Infotech

Driven by Technology