JSF 2.0 tutorials

In this lecture series, i will disucss about JSF2, the lecutre series is divided in below topics, you can easily learn JSF by following tutorial series step by step.

SW tool required:
Java 7.x
Tomcat 7.x
Eclipse Juno
JSF 2.1 reference implemenation : grab jar from *javaserverfaces.java.net/**download**.html*
javax.faces-2.1.17.jar

Topics:

1. Introduction to JSF2.0

What is JSF?

– JavaServer Faces (JSF) is the standard web application framework for Java Enterprise Edition (Java EE)

Why should I use JSF?

– As a Java standard, it has solid industry support
– Growing in usage worldwide
– Strong support in Java IDEs

As with all JEE specification JSF2.0 specification is addresss by JSR-314 that begin its work in May 2007 and defined and released JSF 2.0 July 2009

The JSF API is basically a set of Java interfaces
and abstract classes that define a contract which a Reference Implementation (RI) must fulfill

There are two open source JSF RIs available:
– Oracle RI, code-named "Mojarra"
– Apache MyFaces RI

JSF features

MVC

Implements the Model View Controller (MVC) design pattern

RAD

Rapid Application Development for web applications

UI Components

User Interfaces developed with reusable components
Many component suites available, such as ICEfaces

Render-Kits

Components can render themselves
according to multiple client devices

Validation/Conversion

User input and server

Extensibility

Framework is highly extensible via pluggable architecture
navigation-handler, view-handler, phase-listener,
el-resolver, validators, converters

Internationalization

Views can manifest themselves in different languages

Templating

Facelets provides a powerful templating mechanism

Composite Components

Easily create custom pieces of markup with Facelets

In Nutshell what is JSF 2.0?

JavaServer Faces (JSF) is a user interface (UI) framework for Java web applications

- **UI is constructed from reusable UI components**
- **Simplifies data transfer to and from the UI**
- **UI state can be managed between client request**
- **Provides a simple action model mapping client events to server side application code**
- **Allows custom UI Components to be easily created and reused**

Simplified provide simplified Data Transfer b/w view and model layer

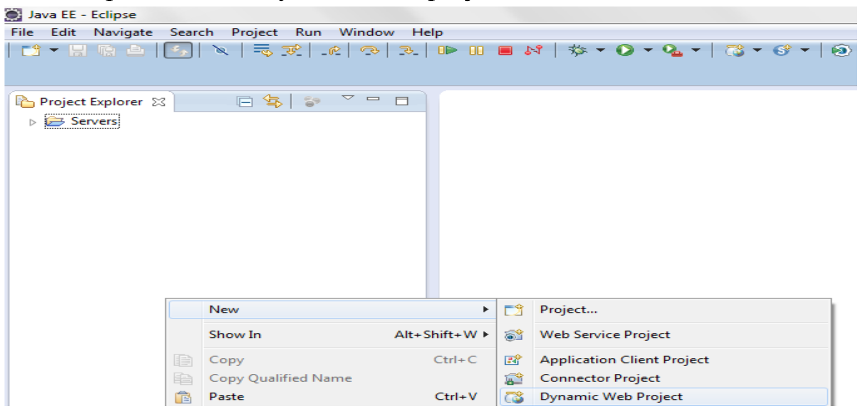
JSF provides a simple value binding mechanism
Simple Java types such as String and int are automatically converted by the framework
There is no need to manually clean and convert values found in the request map
Extension points allow for the conversion of more complex Java objects

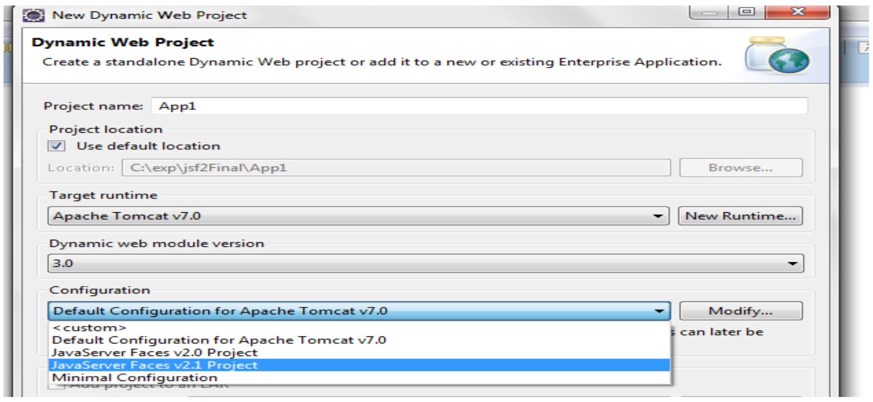
Component Tree

The JSF framework manages the hierarchy in a component tree on the server side. Although components are specified declaratively using XML markup, their run-time representation are Java class instances that are maintained by the JSF framework in a component tree. The componet tree has short life cycle nearly equal to duration of request and response.

Enough theory , lets start hello world JSF application

First configure eclipse with tomcat7
Then
Start eclipse…choose dynamic web project




New Dynamic Web Project

Dynamic Web Project

Create a standalone Dynamic Web project or add it to a new or existing Enterprise Application.

Project name: App1

Project location

☑ Use default location

Location: C:\exp\jsf2Final\App1

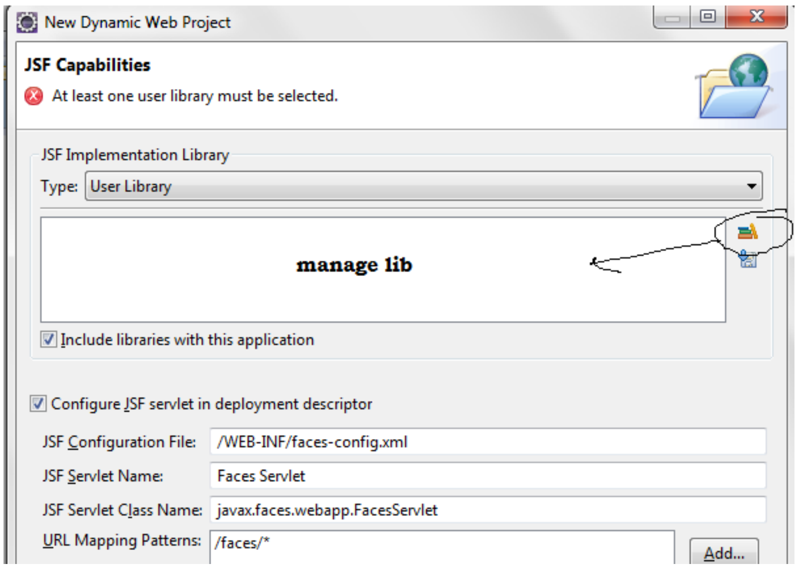
Target runtime: Apache Tomcat v7.0

Dynamic web module version: 3.0

Configuration: Default Configuration for Apache Tomcat v7.0

<custom>
Default Configuration for Apache Tomcat v7.0
JavaServer Faces v2.0 Project
JavaServer Faces v2.1 Project
Minimal Configuration

Choose configuration JSF v2.1 project….click next…


New Dynamic Web Project

JSF Capabilities

At least one user library must be selected.

JSF Implementation Library

Type: User Library

☑ Include libraries with this application

☑ Configure JSF servlet in deployment descriptor

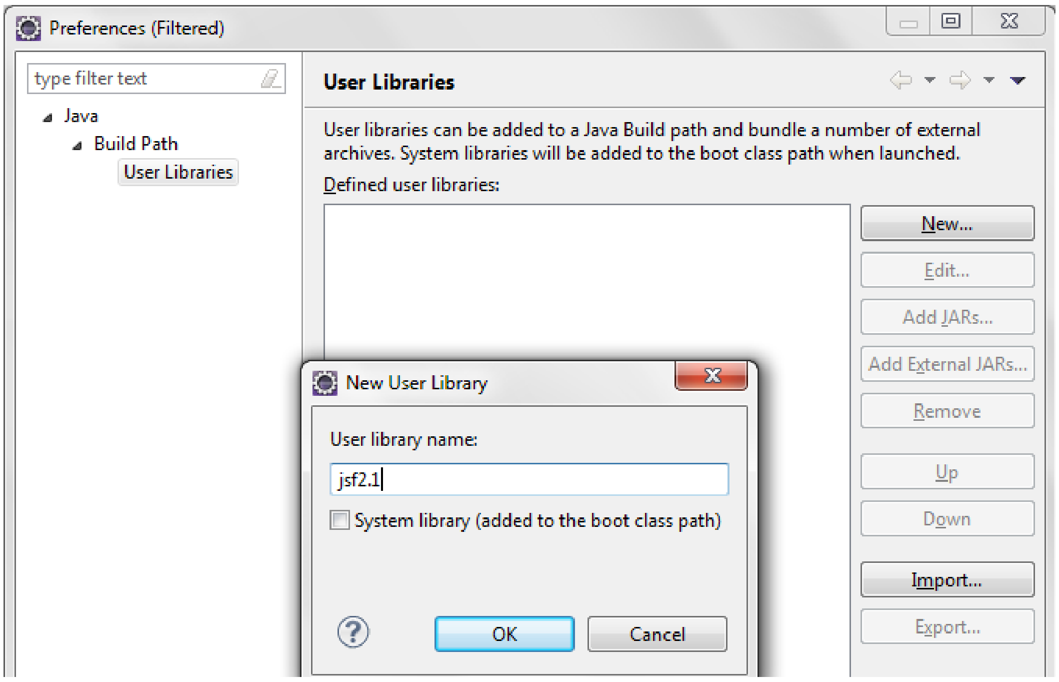
JSF Configuration File: /WEB-INF/faces-config.xml

JSF Servlet Name: Faces Servlet

JSF Servlet Class Name: javax.faces.webapp.FacesServlet

URL Mapping Patterns: /faces/*

Now you can see eclipse is shouting for configuration …
Click manage libery …create user library give suitable name


Preferences (Filtered)

User Libraries

User libraries can be added to a Java Build path and bundle a number of external archives. System libraries will be added to the boot class path when launched.

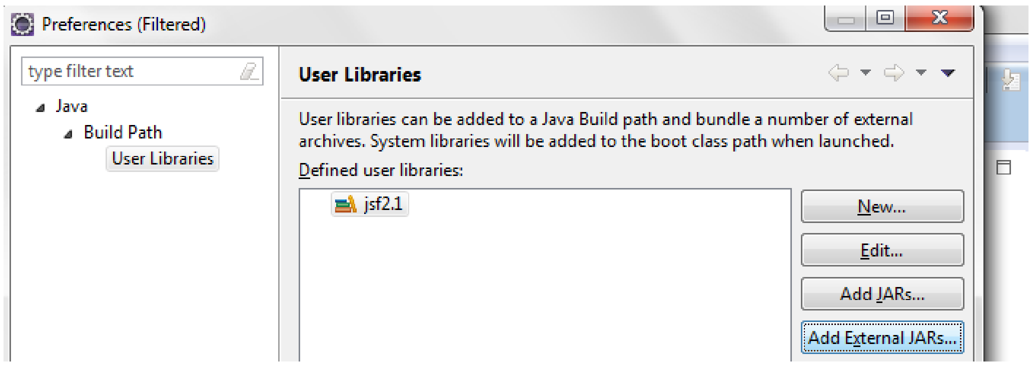
Defined user libraries:

New User Library

User library name: jsf2.1

☐ System library (added to the boot class path)

Now click add external jar and provide jsf2.1 jar downloaded earlier…only one jar is required…

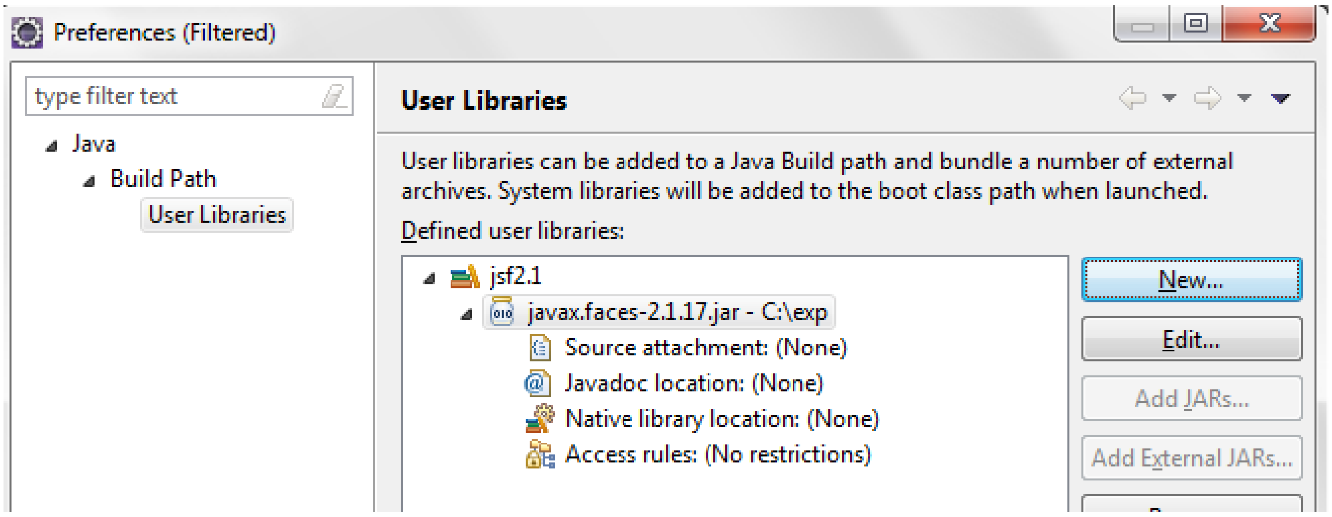

Preferences (Filtered)

User Libraries

User libraries can be added to a Java Build path and bundle a number of external archives. System libraries will be added to the boot class path when launched.

Defined user libraries:

jsf2.1

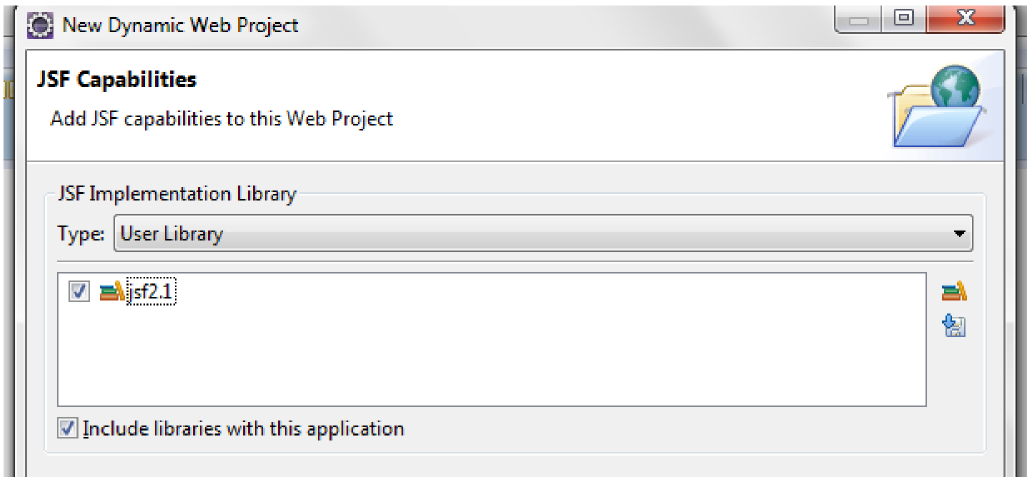

Preferences (Filtered)

User Libraries

User libraries can be added to a Java Build path and bundle a number of external archives. System libraries will be added to the boot class path when launched.

Defined user libraries:

```
jsf2.1
  javax.faces-2.1.17.jar - C:\exp
    Source attachment: (None)
    Javadoc location: (None)
    Native library location: (None)
    Access rules: (No restrictions)
```


New Dynamic Web Project

JSF Capabilities

Add JSF capabilities to this Web Project

JSF Implementation Library

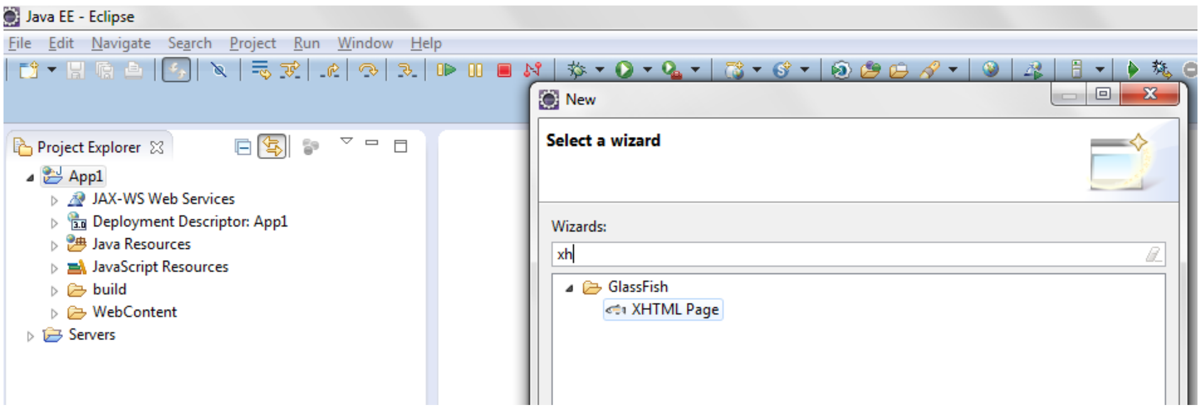
Type: User Library

[✓] jsf2.1

[✓] Include libraries with this application

The say finished

Now create an xhtml page (don’t use jsp for view in JSF2)



Project Explorer

- App1
  - JAX-WS Web Services
  - Deployment Descriptor: App1
  - Java Resources
  - JavaScript Resources
  - build
  - WebContent
- Servers

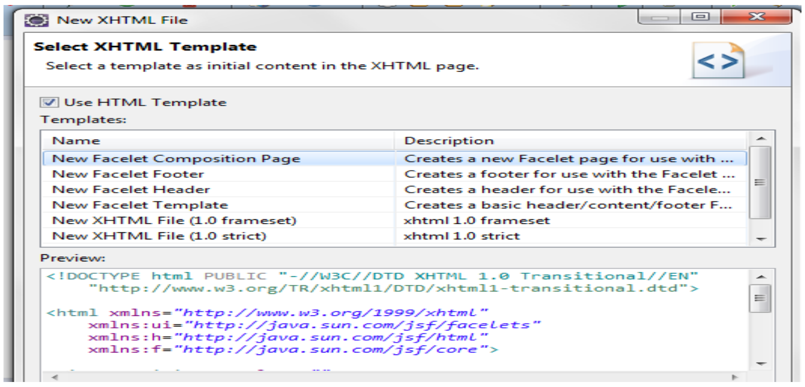
New

Select a wizard

Wizards:

xh

- GlassFish
  - XHTML Page


New XHTML File

Select XHTML Template

Select a template as initial content in the XHTML page.

[✓] Use HTML Template

Templates:

Name	Description
New Facelet Composition Page	Creates a new Facelet page for use with ...
New Facelet Footer	Creates a footer for use with the Facelet ...
New Facelet Header	Creates a header for use with the Facele...
New Facelet Template	Creates a basic header/content/footer F...
New XHTML File (1.0 frameset)	xhtml 1.0 frameset
New XHTML File (1.0 strict)	xhtml 1.0 strict

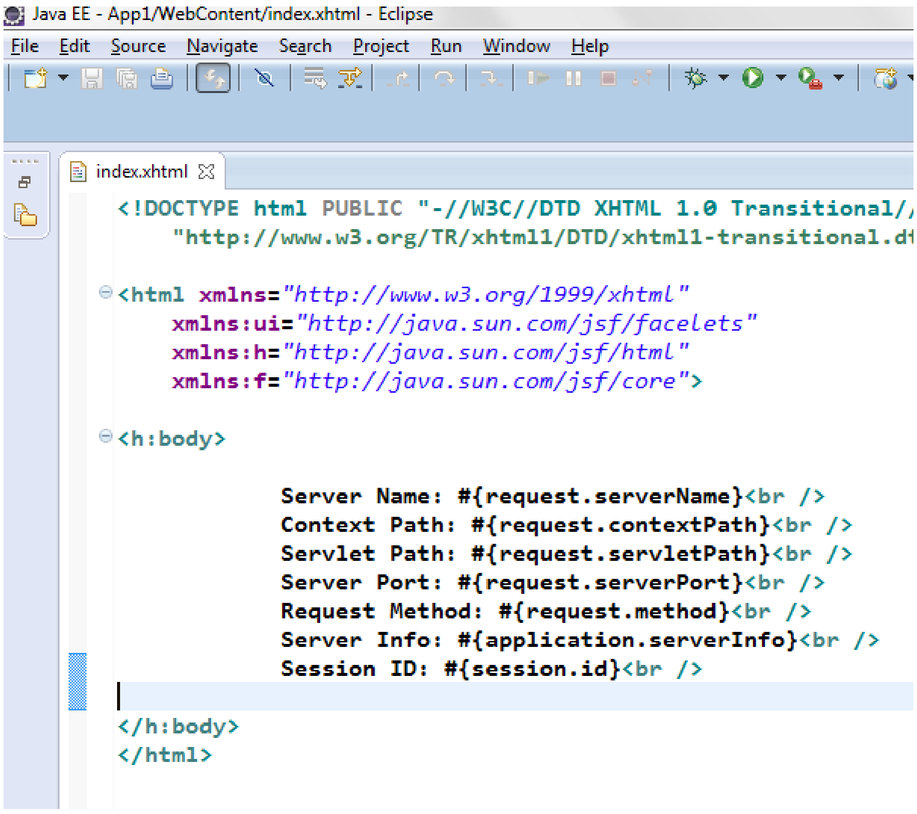
Preview:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">
```

Now add following to created page , to get server, session etc information on index.xhtml

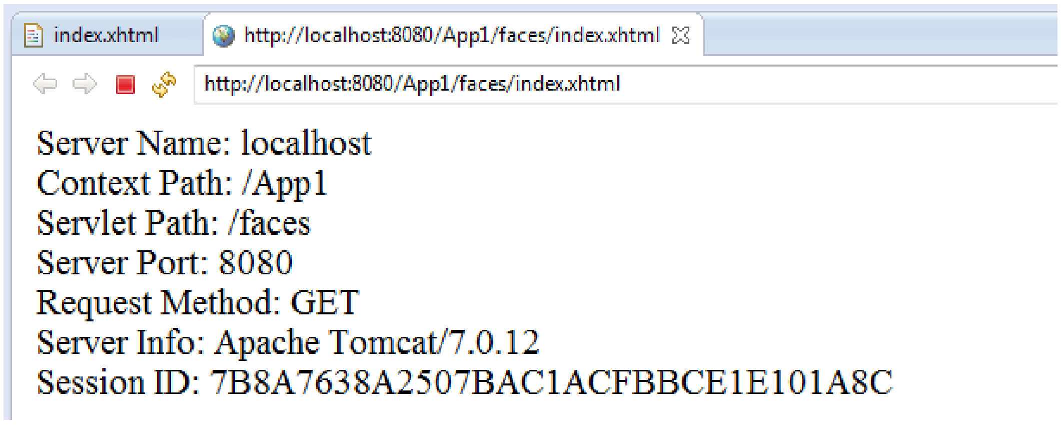
```
Server Name: #{request.serverName}<br />
Context Path: #{request.contextPath}<br />
Servlet Path: #{request.servletPath}<br />
Server Port: #{request.serverPort}<br />
Request Method: #{request.method}<br />
Server Info: #{application.serverInfo}<br />
Session ID: #{session.id}<br />
```

Now index.xhtml looks like this…….


Java EE - App1/WebContent/index.xhtml - Eclipse
File Edit Source Navigate Search Project Run Window Help
index.xhtml

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dt

<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:ui="http://java.sun.com/jsf/facelets"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:f="http://java.sun.com/jsf/core">

<h:body>

            Server Name: #{request.serverName}<br />
            Context Path: #{request.contextPath}<br />
            Servlet Path: #{request.servletPath}<br />
            Server Port: #{request.serverPort}<br />
            Request Method: #{request.method}<br />
            Server Info: #{application.serverInfo}<br />
            Session ID: #{session.id}<br />

</h:body>
</html>
```

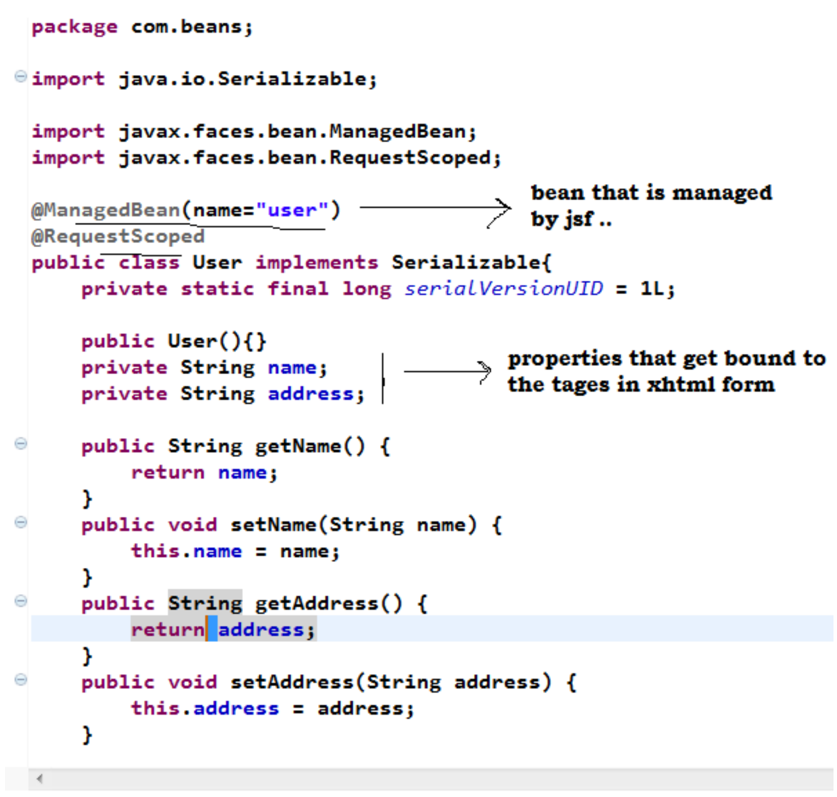
Now run it…get the output……


index.xhtml
http://localhost:8080/App1/faces/index.xhtml
http://localhost:8080/App1/faces/index.xhtml

```
Server Name: localhost
Context Path: /App1
Servlet Path: /faces
Server Port: 8080
Request Method: GET
Server Info: Apache Tomcat/7.0.12
Session ID: 7B8A7638A2507BAC1ACFBBCE1E101A8C
```

Now lets do an simple application that accept user firstName and lastName and echo it…although application is very simple but I use it to demonstrate JSF works?

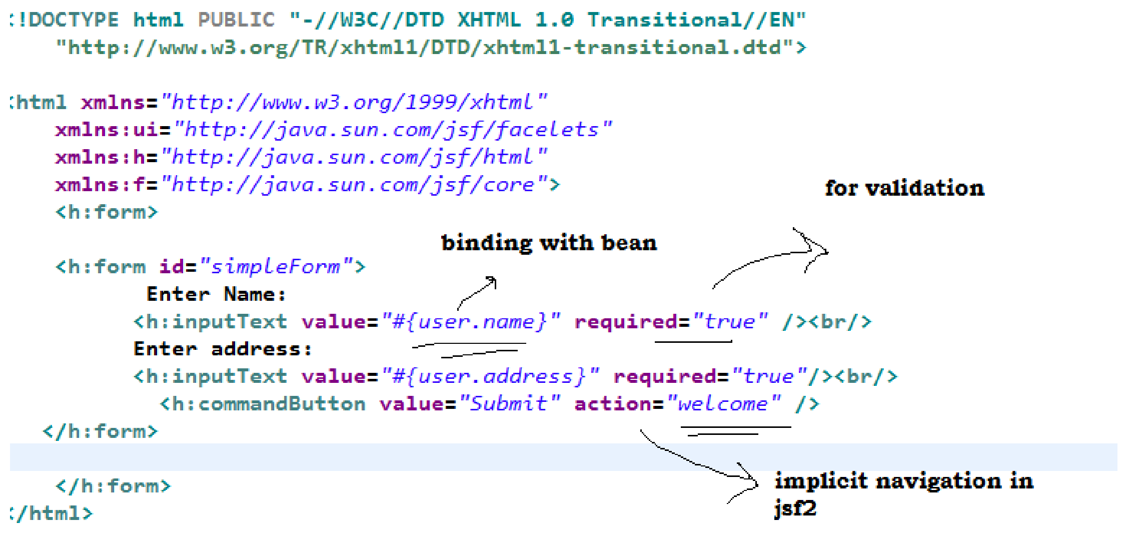
Now first of all create an POJO User as shown…


bean that is managed by jsf ..
properties that get bound to the tages in xhtml form

```java
package com.beans;

import java.io.Serializable;

import javax.faces.bean.ManagedBean;
import javax.faces.bean.RequestScoped;

@ManagedBean(name="user")
@RequestScoped
public class User implements Serializable{
    private static final long serialVersionUID = 1L;

    public User(){}
    private String name;
    private String address;

    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getAddress() {
        return address;
    }
    public void setAddress(String address) {
        this.address = address;
    }
```

Now copy following in index.xhtml

```xml
<h:form id="simpleForm">
      Enter Name:
     <h:inputText value="#{user.name}" required="true" /><br/>
     Enter address:
     <h:inputText value="#{user.address}" required="true"/><br/>
      <h:commandButton value="Submit" action="welcome" />
 </h:form>
```

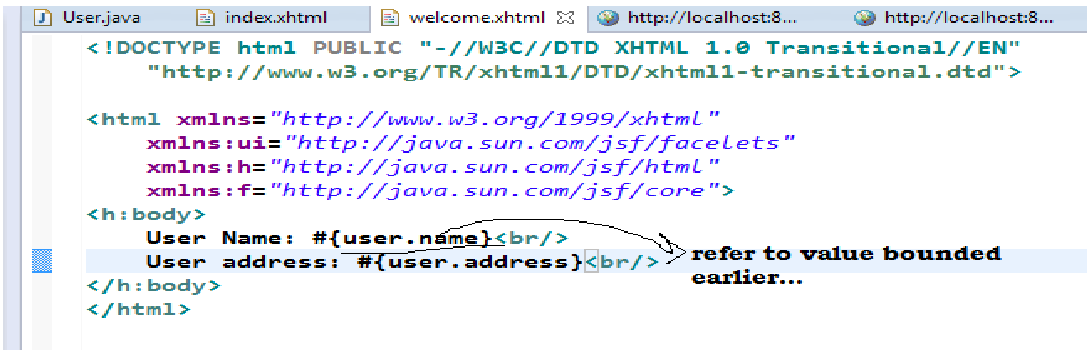


```xml
:!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

:html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:ui="http://java.sun.com/jsf/facelets"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:f="http://java.sun.com/jsf/core">
    <h:form>

    <h:form id="simpleForm">
          Enter Name:
         <h:inputText value="#{user.name}" required="true" /><br/>
         Enter address:
         <h:inputText value="#{user.address}" required="true"/><br/>
           <h:commandButton value="Submit" action="welcome" />
   </h:form>

    </h:form>
:/html>
```

When user fills values in this form value is bound to an instance of bean (ie in request scope), when user submit the form action mentioned is “welcome” what it means?

It means that control goes to welcome.xhtml (aka RequestDispaching in servlet/jsp)

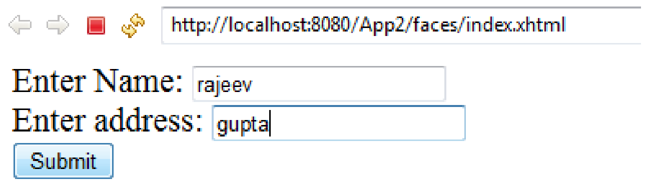
Now write welcome.xhtml



```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:ui="http://java.sun.com/jsf/facelets"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:f="http://java.sun.com/jsf/core">
<h:body>
    User Name: #{user.name}<br/>
    User address: #{user.address}<br/>
</h:body>
</html>
```

refer to value bounded earlier...

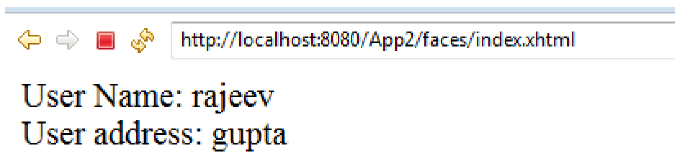
Now run the application



http://localhost:8080/App2/faces/index.xhtml

Enter Name: rajeev

Enter address: gupta



http://localhost:8080/App2/faces/index.xhtml

User Name: rajeev

User address: gupta

2. Managed bean and Scoping

Although We have already used managed bean in last example but now will dicuss it in detail specially about scoping.

Managed bean?

- JSF allows users to design a complex tree of named POJO Beans
- Beans have a predetermined lifespan known as scope
- Beans can be defined using:
  - Managed Bean Creation facility (faces-config.xml)
  - Java EE 5 annotations
- Managed Beans also have a lifecycle which depends on their specified scope

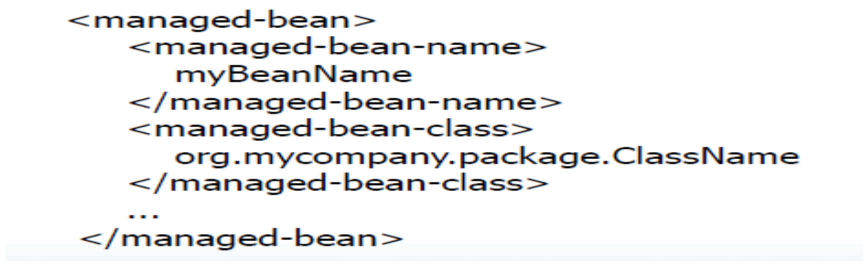
Bean Names

- JSF 2 introduces the @ManagedBean annotation
- Names can be specified with the name attribute
  - @ManagedBean(name="someName")
- If a name is not specified the class name is used as the Bean name, mixed case starting with lower case
- The eager attribute can be used to insure that a bean is loaded in a non-lazy fashion
  - @ManagedBean(name="someName", eager=true)
- Note: import javax.faces.bean. ManagedBean;

Bean Scopes

JSF 1.x originally defined four scope types:
Application
Lifespan continues as long as the web application is deployed
Session
Lifespan of the HttpSession, destroyed by session timeout or manual invalidation
Unique to each user but share across multiple browser tabs
Request
Lifespan duration of an HTTP request received by the server and response sent to client
No Scope
Bean isn't placed into scope

JSF 2 introduces three new scopes:
View
Bean lasts the duration of the view
Page navigation or page refreshes cause the Bean to be destroyed and reinitialized
Flash
Short conversation-style scope that exists for a single view transition including reloads
Custom
Allows developers to implement their own custom scope behavior
Note: beans can be configured in in faces-config.xml… but not requied in JSF2



```
<managed-bean>
    <managed-bean-name>
       myBeanName
    </managed-bean-name>
    <managed-bean-class>
       org.mycompany.package.ClassName
    </managed-bean-class>
    ...
 </managed-bean>
```

Important scopes are :

Application: one per application
Session :one per browser,useful for session mgt
Request : per http request/response cycle
View :Bean lasts the duration of the view
Page navigation or page refreshes cause the Bean to be destroyed and reinitialized

Scopes can also be defined with annotations
– @ApplicationScoped
– @SessionScoped
– @ViewScoped
– @RequestScoped
– @CustomScoped(value="#{someMap}")
– @NoneScoped

Note: make sure to import:
javax.faces.bean.Xscoped;

<u>Now lets demostrate an example that highlight effect of bean scoping and when to use what</u>

Create 4 beans as shown:

```
@ManagedBean(name="applicationScopeBean")
@ApplicationScoped
public class ApplicationScopeBean {
        @Override
        public String toString(){
                return "ApplicationScopeBean:"+super.toString();
        }
}
```

```
import javax.faces.bean.ManagedBean;
import javax.faces.bean.RequestScoped;

@ManagedBean
@RequestScoped
public class RequestScopeBean {

        @Override
        public String toString(){
                return "RequestScopeBean:"+super.toString();
        }
}
```

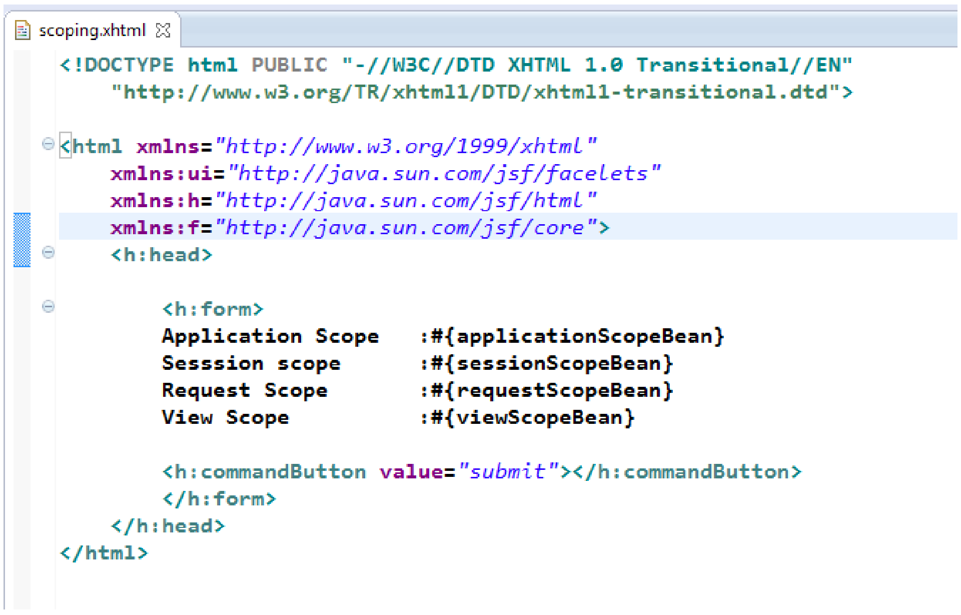
```
import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;

@ManagedBean
@SessionScoped
public class SessionScopeBean {

	@Override
	public String toString(){
		return "SessionScopeBean:"+super.toString();
	}

}
```

```
import javax.faces.bean.ManagedBean;
import javax.faces.bean.ViewScoped;

@ManagedBean(name="viewScopeBean")
@ViewScoped
public class ViewScopeBean {

	@Override
	public String toString(){
		return "ViewScopeBean:"+super.toString();
	}
}
```

As you can see code is very simple, code will provide object stamp.

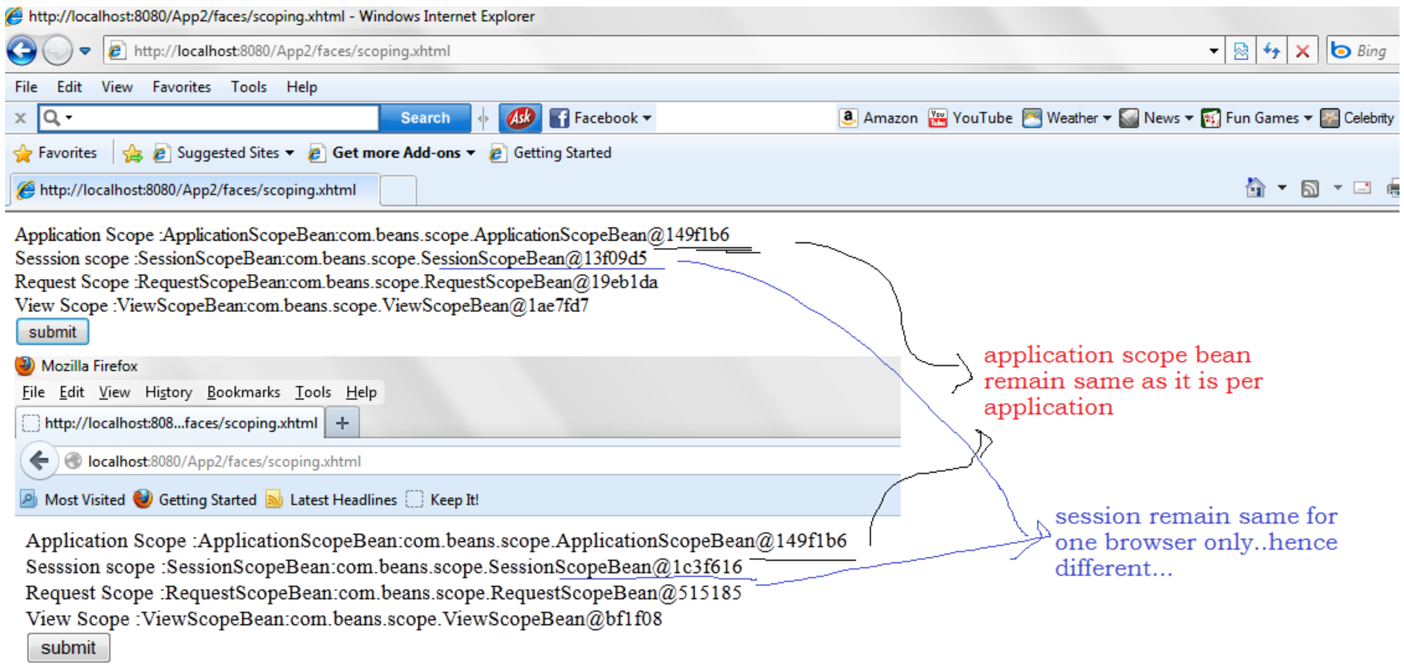
We will print object stemp for all bean using………

```
<h:form>
		Application Scope		:#{applicationScopeBean}<br/>
		Sesssion scope			:#{sessionScopeBean}<br/>
		Request Scope			:#{requestScopeBean}<br/>
		View Scope				:#{viewScopeBean}<br/>

<h:commandButton value="submit"></h:commandButton>
</h:form>
```



```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:ui="http://java.sun.com/jsf/facelets"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:f="http://java.sun.com/jsf/core">
    <h:head>

        <h:form>
        Application Scope    :#{applicationScopeBean}
        Sesssion scope       :#{sessionScopeBean}
        Request Scope        :#{requestScopeBean}
        View Scope           :#{viewScopeBean}

        <h:commandButton value="submit"></h:commandButton>
        </h:form>
    </h:head>
</html>
```

Now run this file on different browse


http://localhost:8080/App2/faces/scoping.xhtml - Windows Internet Explorer

Application Scope :ApplicationScopeBean:com.beans.scope.ApplicationScopeBean@149f1b6
Sesssion scope :SessionScopeBean:com.beans.scope.SessionScopeBean@13f09d5
Request Scope :RequestScopeBean:com.beans.scope.RequestScopeBean@19eb1da
View Scope :ViewScopeBean:com.beans.scope.ViewScopeBean@1ae7fd7

Mozilla Firefox

Application Scope :ApplicationScopeBean:com.beans.scope.ApplicationScopeBean@149f1b6
Sesssion scope :SessionScopeBean:com.beans.scope.SessionScopeBean@1c3f616
Request Scope :RequestScopeBean:com.beans.scope.RequestScopeBean@515185
View Scope :ViewScopeBean:com.beans.scope.ViewScopeBean@bf1f08

application scope bean remain same as it is per application

session remain same for one browser only..hence different...

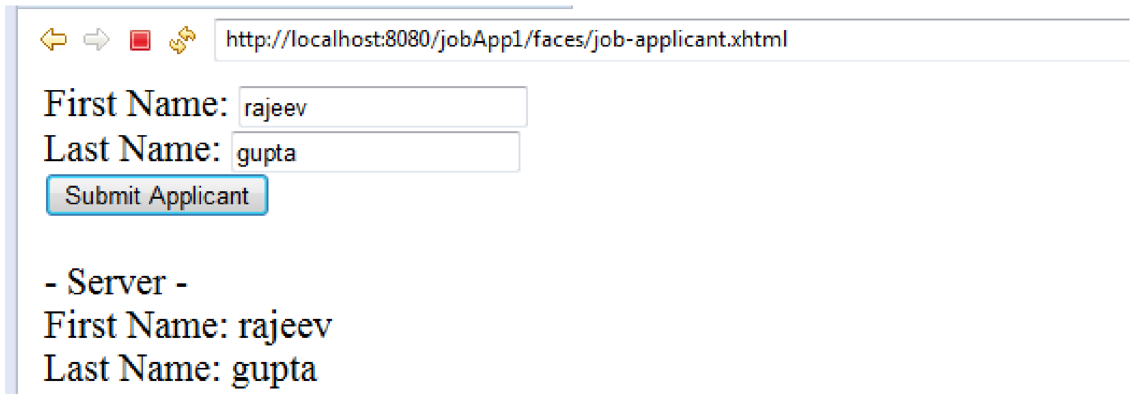
Now run it on same browser but on different tab..what is your observation.
Session stemp remain same.
But request and view stemp changes

Now question is what is the differnce b/w request and view scoped bean?

View bean remain same for an particular view , but request scope is only for one cycle of request/resonse and hence smallest in scope among four we discussed.

Let us now create an new application jobApplicant

This application is going to be used in following excercies


http://localhost:8080/jobApp1/faces/job-applicant.xhtml

First Name: rajeev
Last Name: gupta

- Server -
First Name: rajeev
Last Name: gupta

Job-applicant.xhtml

```
<h:body>
        <h:form>
                <h:outputLabel for="firstName" value="First Name: " />
                <h:inputText id="firstName" value="#{jobApplicant.firstName}" />
                <br />
                <h:outputLabel for="lastName" value="Last Name: " />
                <h:inputText id="lastName" value="#{jobApplicant.lastName}" />
                <br />
```

```
                    <h:commandButton value="Submit Applicant" />
            </h:form>
            <br />
    - Server -<br />
    First Name: <h:outputText value="#{jobApplicant.firstName}" />
            <br />
    Last Name: <h:outputText value="#{jobApplicant.lastName}" />
            <br />
</h:body>
```

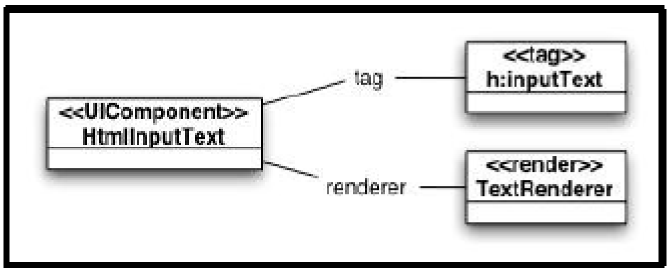
```
package com.beans;

import javax.faces.bean.ManagedBean;
import javax.faces.bean.RequestScoped;

@ManagedBean(name="jobApplicant")
@RequestScoped
public class JobApplicant {

            private String firstName;
            private String lastName;

            public JobApplicant(){}

            //getter and setter

}
```

3. JSF tags

Input-output tags, Selection tags, Control tags, data-table

JSF Input and Output Components

There are 3 parts to a component:
– The Component Class
– The Tag Class
– The Renderer Class



JSF Component Architecture

– The default JSF components render html (h: namespace)
– More information on component development will be

covered in later lectures
JSF

JSF input component

JSF contains a handful of basic components that are
equivalent to the HTML 4 elements
– <h:form />
– <h:inputText />
– <h:inputTextarea />
– <h:inputSecret />
– <h:inputHidden />
– <h:outputLabel />
– <h:outputLink />
– <h:outputFormat />
– <h:outputText />

Component Tags

Component tags are placed on JSF pages/views:

<h:outputText />
Tag attributes allow us to customize the appearance and behavior of components:
<h:outputText value="Hello World" rendered="true" />

Tags are nested in a parent-child containment format:

```
<h:form>
<h:outputLabel for="fullName" />
<h:inputText id="fullName" />
</h:form>
```

Value Binding

Bean values are assigned to component attributes using JSF EL syntax
• For example the managed bean myBean's instance variable 'value1' is assigned to the input component as follows:

```
<h:inputText value="#{myBean.value1}"/>
<h:inputText value="#{myBean.value1}"
rendered="#{myBean.value2}"/>
```

Input Value Binding

Input components in JSF require that the bound bean property is mutable :

```
public void setValue1(Object value1){...}
public Object getValue1(){...};
```

• If the JSF introspection mechanism can't find the corresponding setter or getter a run time error will occur
• All other non "value" attribute bindings can be immutable as the setter method is never called by the JSF framework

Output Value Binding

Output components in JSF assume that the associated value bindings are immutable but it is not a requirement

```
public Object getValue1(){...};
```

- If the JSF introspection mechanism can't find the corresponding getter a run time error will occur
- All non "value" component attribute bindings can be immutable as the setter method is never called by the JSF framework

Form Input Component

The form input component is a required parent tag for all input components
• Any input components in the form tag will be submitted to the server when submit occurs

```
<h:form />
<h:inputText value="#{bean.userName}"/>
...
</h:form/>
```

• Form tags can not be embedded but there can be more then one form per page

InputText Component

Input Text is the same as the <input type="text"/> in html 4 allowing client side users to input text

- The value binding can be of type String, Number and all number primitives. JSF takes care of conversion
- InputText can be quite powerful when combined with converters, validators and ajax tags which will be explained in more detail later

InputTextArea Component

InputTextArea is the same as the <input type="textarea"/> in html 4 allowing client side users to input text

• The value binding should be of the type String for the value attribute

InputSecret Component

InputSecret Text is the same as the
<input type="password"/> in HTML 4 allowing client
to enter hidden or secret text

• The component attribute autocomplete="off" is handy for suppressing the form auto complete feature of most modern browsers

InputHidden Component

Similar to the <input type="hidden"/> in HTML 4
• Allows JSF developers to include hidden form data that will be submitted with the other form elements • Not used as often in JSF as in standard HTML as

Bean scopes provide more intuitive state saving

OutputLabel Component

Renders the same output as the HTML 4 <label> tag

- Generally used in combination with an input component
- When the id attribute of an input component matches the for attribute of an outputLabel a fieldset tag will automatically be inserted by the framework

OutputLink Component

Renders the same output as the HTML 4 <a> tag

- Not commonly used in JSF as most developers use framework features that aren't implicitly supported by this component
- JSF 2.0 introduces <h:link /> component which allows developers to use HTTP GET submits instead

of the standard JSF POST submits

OutputFormat Component

Allows developers to use Java i18n message bundles that have specified input parameters
• This component will be covered in later lectures around message bundles

• A simple example of its usage:
<h:outputFormat value="Line number {0} !">
<f:param value="153,44"/>
</h:outputFormat>

OutputText Component

Renders the same output as the HTML 4 <span> tag
• JSF 2.0 EL notation has somewhat reduced the use of the outputText component

#{myBean.value1}

Equivalent to

<h:outputText value="#{myBean.value1}" />

• However it is still used when JSF conversion is needed

Command Components

The h:commandButton and h:commandLink components Implement UICommand

• JSF provides two ways to detect when a user interacts with UICommand components:
  – Add the actionListener attribute
  – Add the action attribute

• ActionListener and Action methods are usually located in a controller bean, not a model bean

• During the Invoke Application Phase of the lifecycle ActionListeners are called first followed by Actions

ActionListener and Action

Actions are generally used to invoke navigation
• ActionListeners are primarily used to execute business logic that does not result in navigation

• Both attributes can be used on a component:
<h:commandButton actionListener="#{controller.authenticate}" action="#{controller.login}" value="login" />

• The 'authenticate' method is called first in the lifecyclefollowed by the 'login' method

commandButton

The h:commandButton renders an HTML <input type="button" /> tag

• Unlike the input tag, this component will invoke an actionListener and/or action method when clicked
    – This is a fundamental feature of an action based framework.

• The image attribute tells the component that it should render a specified image rather then the default button widget

commandLink

The h:commandLink renders an HTML <a/> tag

• Unlike the <a/> tag this component will invoke an actionListener and/or action method when clicked
    – This is a fundamental feature of an action based framework

• Child elements of the commandLink tag are wrapped by the anchor tag's functionality

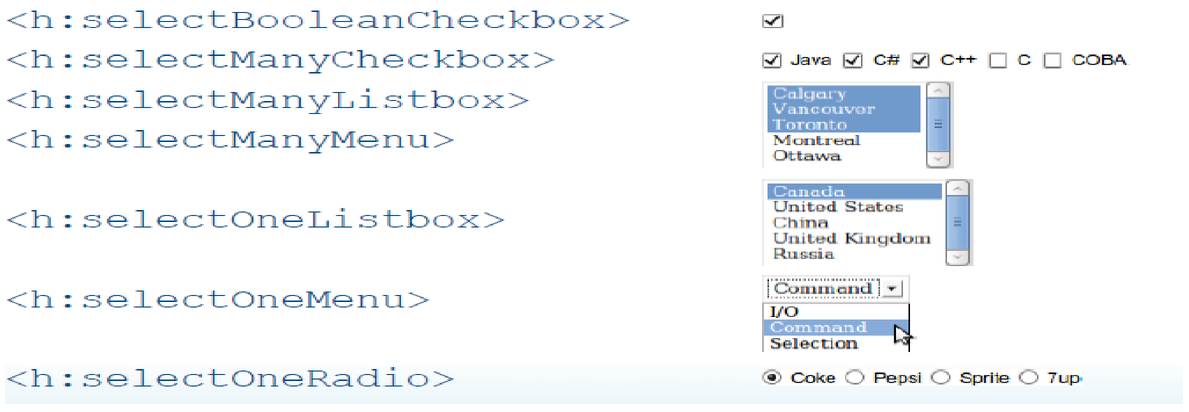
Immediate & Command Components

Command Components execute in the Invoke Application phase of the JSF lifecycle

> • Conversion/Validation errors encountered in the Process Validations phase will cause the lifecycle to skip the Invoke Application Phase
> • The 'Immediate' attribute can be used to move the execution of ActionListeners and Actions to the Apply Request Values phase
> • The 'Immediate' attribute would be used with a "Cancel" button to insure ActionListeners and Actions are called

<u>JSF Selection Components</u>

JSF contains a handful of basic "selection" components that are equivalent to the HTML 4 select elements.



```
<h:selectBooleanCheckbox>
<h:selectManyCheckbox>
<h:selectManyListbox>
<h:selectManyMenu>

<h:selectOneListbox>

<h:selectOneMenu>

<h:selectOneRadio>
```

Basic Selection Tag Usage

Each selection tag works like a parent-child container, and
has two necessary parts

The parent tag with the currently selected value:
```
<h:selectOneMenu value="#{modelBean.ourColor}">
```

The child tag(s) listing available items:
```
<f:selectItem itemLabel="Red" itemValue="red" />
```

The finished selectOneMenu code would be:
```
<h:selectOneMenu value="#{modelBean.ourColor}">
<f:selectItem itemLabel="Red" itemValue="red" />
<f:selectItem itemLabel="Green" itemValue="green" />
<f:selectItem itemLabel="Blue" itemValue="blue" />
</h:selectOneMenu>
```

Dynamic Selection Tag Usage

The available items can also be dynamically pulled from a bean:
```
<h:selectOneMenu value="#{modelBean.ourColor}">
        <f:selectItems value="#{modelBean.availableColors}" />
</h:selectOneMenu>
```

The dynamic items could be an Array of Strings:
```
        private String[] availableColors = {"Red", "Blue", "Green"};
        public String[] getAvailableColors() {
        return availableColors;
        }
```

The items can now be modified directly in the backing bean list instead of at the page level

Now We will continue our previous program job applicant application

Now we will add two selection components to our page:
  – h:selectOneRadio for title
• With a series of hardcoded f:selectItem
  – h:selectOneMenu for country
  • With a bound list of f:selectItems

• We will also introduce an h:panelGrid component to clean up our form layout

Step 1: Insert h:selectOneRadio Component

Insert the following at the top of the form:

```
	<h:outputLabel for="title" value="Title: " />
	<h:selectOneRadio id="title" value="#{jobApplicant.title}">
	<f:selectItem itemLabel="Dr." itemValue="1" />
	<f:selectItem itemLabel="Ms." itemValue="2" />
	<f:selectItem itemLabel="Mrs." itemValue="3" />
	<f:selectItem itemLabel="Miss" itemValue="4"/>
	<f:selectItem itemLabel="Mr." itemValue="5"/>
	</h:selectOneRadio>
```

• Each f:selectItem is hard coded into the page which is not a best practice
  – Have to copy-paste between pages using the same component
  – Incorrectly mixes model and view
  – Better to dynamically load from a bean or database

Step 2: Insert h:selectOneMenu Component

Insert the following markup before the h:commandButton:

```
<h:outputLabel for="country" value="Country: " />
<h:selectOneMenu id="country" value="#{jobApplicant.country}">
<f:selectItem itemLabel="-Select-" noSelectionOption="true" />
<f:selectItems value="#{countryList.countries}" />
</h:selectOneMenu>
<br />
```

• Note our combination of selectable items:
  – f:selectItem is used to present a default String that is not considered a selection (via noSelectionOption)
  – f:selectItems has a value binding

• The values are not hard coded into the page and can be dynamically updated

Step 3: Add Variables to JobApplicant
Add two variables to JobApplicant.java:

```
private String title;
private String country;
```

• Generate getters and setters

• Note: These two variables are used to bind the model to the view

Step 4: Create CountryList Bean
Create a new bean class CountryList

```
@ManagedBean
@ApplicationScoped
public class CountryList {
	private String[] countries = { "Canada", "United States" };

	public String[] getCountries() {
```

```
                return countries;
        }

        public void setCountries(String[] countryList) {
                this.countries = countryList;
        }
}
```

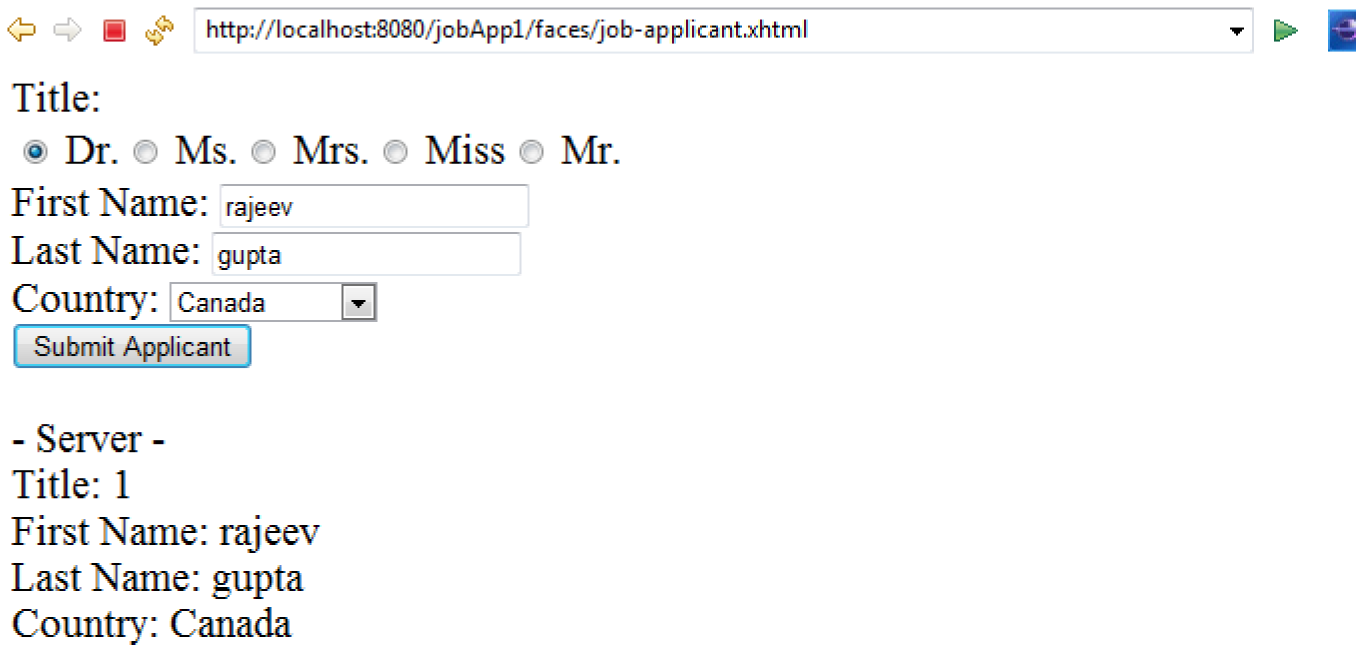
• Note: ApplicationScoped because it serves as a common list of countries bound to the h:selectOneMenu (Support Managed Bean)

Step 5: Update Server Output

Update the server output after h:form tag:

```
<br />
- Server -<br />
Title: <h:outputText value="#{jobApplicant.title}" /><br />
First Name: <h:outputText value="#{jobApplicant.firstName}"
/><br />
Last Name: <h:outputText value="#{jobApplicant.lastName}"
/><br />
Country: <h:outputText value="#{jobApplicant.country}"
/><br />
```

Now run it to have this



Title:
[● Dr.] ○ Ms. ○ Mrs. ○ Miss ○ Mr.
First Name: rajeev
Last Name: gupta
Country: Canada

- Server -
Title: 1
First Name: rajeev
Last Name: gupta
Country: Canada

h:panelGrid Description

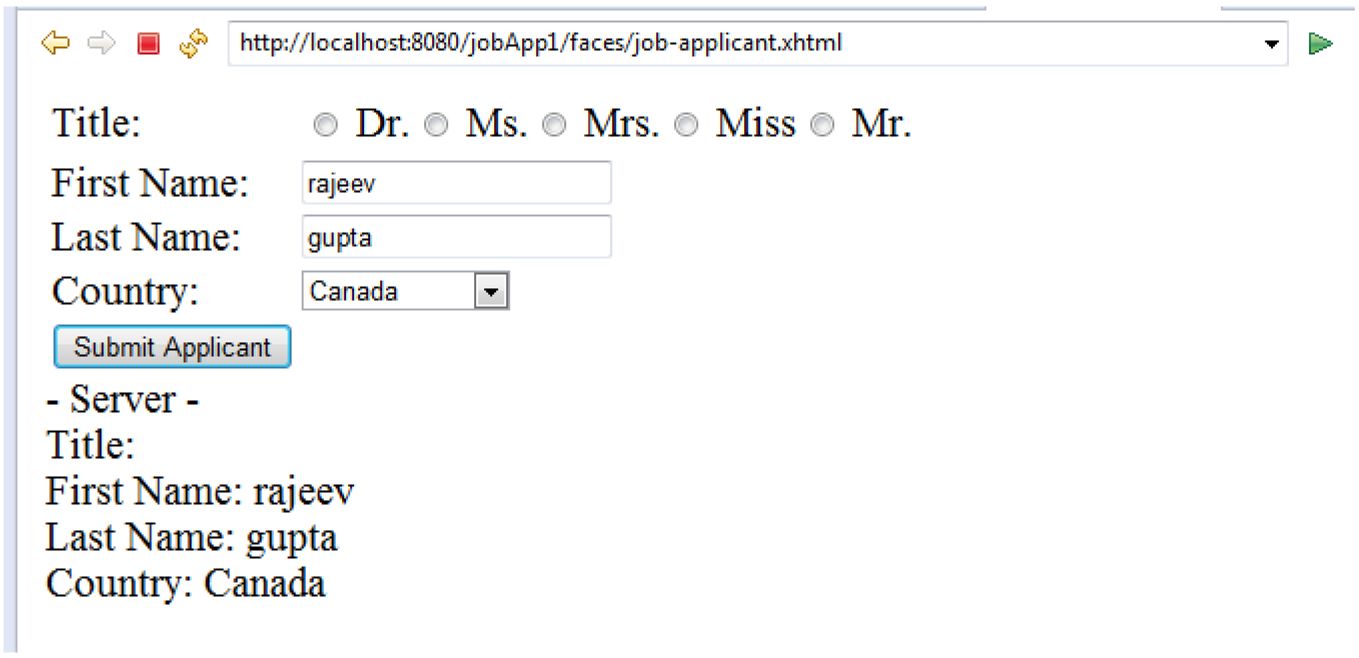
h:panelGrid renders an HTML table
• Will be used to layout our existing components
• New rows are defined by the integer attribute 'columns'
– Once X number of child components are rendered a new row is started
• Child components are each placed in a table cell

Step 6: Use h:panelGrid for Layout

Add panelGrid to job-applicant.xhtml (remove br tags):

```
<h:form>
            <h:panelGrid columns="2">
                  <h:outputLabel for="title" value="Title: " />
                  <h:selectOneRadio id="title" value="#{jobApplicant.title}">
                        <f:selectItem itemLabel="Dr." itemValue="1" />
                        <f:selectItem itemLabel="Ms." itemValue="2" />
                        <f:selectItem itemLabel="Mrs." itemValue="3" />
                        <f:selectItem itemLabel="Miss" itemValue="4" />
                        <f:selectItem itemLabel="Mr." itemValue="5" />
                  </h:selectOneRadio>

                  <h:outputLabel for="firstName" value="First Name: " />
                  <h:inputText id="firstName" value="#{jobApplicant.firstName}" />

                  <h:outputLabel for="lastName" value="Last Name: " />
                  <h:inputText id="lastName" value="#{jobApplicant.lastName}" />


                  <h:outputLabel for="country" value="Country: " />
                  <h:selectOneMenu id="country" value="#{jobApplicant.country}">
                        <f:selectItem itemLabel="-Select-" noSelectionOption="true" />
                        <f:selectItems value="#{countryList.countries}" />
                  </h:selectOneMenu>
                  <h:commandButton value="Submit Applicant" />
            </h:panelGrid>
      </h:form>
```

Step 7: Run Application



http://localhost:8080/jobApp1/faces/job-applicant.xhtml

Title: ○ Dr. ○ Ms. ○ Mrs. ○ Miss ○ Mr.
First Name: rajeev
Last Name: gupta
Country: Canada
Submit Applicant

- Server -
Title:
First Name: rajeev
Last Name: gupta
Country: Canada

DataTable

Now we will display data using data table,it is very important to display an arraylist or maps container data from database to create an dynamic table

Consider

```
public class Employee {

        private String id;
        private String name;
        private String phone;
        private Date dob;
        //getter and setter
}

@ManagedBean(name="employeeList")
@RequestScoped
public class EmployeeList {

        public List<Employee>getEmployees(){
                ArrayList<Employee>list=new ArrayList<Employee>();
                Employee e1=new Employee();
                e1.setId("121");
                e1.setName("foo");
                e1.setPhone("34343434");
                e1.setDob(new ~~Date~~(1983,10,2));
                Employee e2=new Employee();
                e2.setId("126");
                e2.setName("bar");
                e2.setPhone("543488888888434");
                e1.setDob(new ~~Date~~(1963,7,2));

                Employee e3=new Employee();
                e3.setId("1261");
                e3.setName("jar");
                e3.setPhone("000004343434");
                e1.setDob(new ~~Date~~(1953,1,2));

                list.add(e1);

                list.add(e2);

                list.add(e3);

                return  list;
        }
}
```

Now question is that how to display Employee list in an xhtml page?

Here we are going to use datatable as :

```
<h:dataTable value="#{employeeList.employees}" var="c" border="2">
                <h:column>
                <f:facet name="header">Employee Id:</f:facet>
                        #{c.id}
                </h:column>

                <h:column>
                <f:facet name="header">Employee Name:</f:facet>
                        #{c.name}
```

```
            </h:column>

            <h:column>
            <f:facet name="header">Employee Phone:</f:facet>
                        #{c.phone}
            </h:column>

            <h:column>
            <f:facet name="header">Employee DOB:</f:facet>
            <h:outputText value="#{c.dob}">
                        <f:convertDateTime pattern="dd/mm/yyyy"></f:convertDateTime>
            </h:outputText>

            </h:column>
</h:dataTable>
```

4. Nevigation

In JSF 1.2 we have to explicitly mention nevigation in faces-config.xml file as;

```
<navigation-rule>
  <from-view-id>page1.xhtml</from-view-id>
  <navigation-case>
     <from-outcome>page2</from-outcome>
     <to-view-id>/page2.xhtml</to-view-id>
  </navigation-case>
</navigation-rule>
```

This sort of explicit navigation is not required in JSF 2.0

We have Implicit navigation

How to use it?

1. Outcome in JSF page

```
<h:form>
   <h:commandButton action="page2" value="Move to page2.xhtml" />
</h:form>
```

2. Outcome in Managed Bean

```
@ManagedBean
@SessionScoped
public class PageController implements Serializable {

   public String moveToPage2(){
      return "page2"; //outcome
   }
}
```

Now code in page1.xhtml

```
<h:form>
   <h:commandButton action="#{pageController.moveToPage2}"
      value="Move to page2.xhtml by managed bean" />
</h:form>
```

Redirection

```
<h:form>
   <h:commandButton action="page2?faces-redirect=true" value="Move to page2.xhtml" />
</h:form>
```

Conditional navigation in JSF2.0

JSF 2 comes with a very flexible conditional navigation rule to solve the complex page navigation flow, see the following conditional navigation rule example

. JSF Page

A simple JSF page, with a button to move from this page to the payment page.

start.xhtml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html">

    <h:body>
                <h2>This is start.xhtml</h2>
                <h:form>
                   <h:commandButton action="payment" value="Payment" />
                </h:form>
    </h:body>
</html>
```

2. Managed Bean

A managed bean to provide sample data to perform the conditional checking in the navigation rule.

```
import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;
import java.io.Serializable;

@ManagedBean
@SessionScoped
public class PaymentController implements Serializable {

                private static final long serialVersionUID = 1L;

                public boolean registerCompleted = true;
                public int orderQty = 99;

                //getter and setter methods
}
```

3. Conditional Navigation Rule

Normally, you declared the simple navigation rule in the “faces-config.xml” like this :

```
<navigation-rule>
                <from-view-id>start.xhtml</from-view-id>
                <navigation-case>
                                <from-outcome>payment</from-outcome>
                                <to-view-id>payment.xhtml</to-view-id>
                </navigation-case>
</navigation-rule>
```

With JSF 2, you can add some conditional checking before it move to the payment page, see following :

faces-config.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<faces-config
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-facesconfig_2_0.xsd"
    version="2.0">

                <navigation-rule>
                                <from-view-id>start.xhtml</from-view-id>
                                <navigation-case>
```

```
                        <from-outcome>payment</from-outcome>
                        <if>#{paymentController.orderQty &lt; 100}</if>
                        <to-view-id>ordermore.xhtml</to-view-id>
                </navigation-case>
                <navigation-case>
                        <from-outcome>payment</from-outcome>
                        <if>#{paymentController.registerCompleted}</if>
                        <to-view-id>payment.xhtml</to-view-id>
                </navigation-case>
                <navigation-case>
                        <from-outcome>payment</from-outcome>
                        <to-view-id>register.xhtml</to-view-id>
                </navigation-case>
        </navigation-rule>

</faces-config>
```

This is equal to the following Java code :

```
if (from-view-id == "start.xhtml"){

  if(from-outcome == "payment"){

    if(paymentController.orderQty < 100){
            return "ordermore";
    }else if(paymentController.registerCompleted){
            return "payment";
    }else{
            return "register";
    }

  }

}
```

The code should be self explanatory enough.

Note
In the conditional navigation rule, the sequence of the navigation rule does affect the navigation flow, always put the highest checking priority in the top.

5. JSF life cycle and PhaseListners

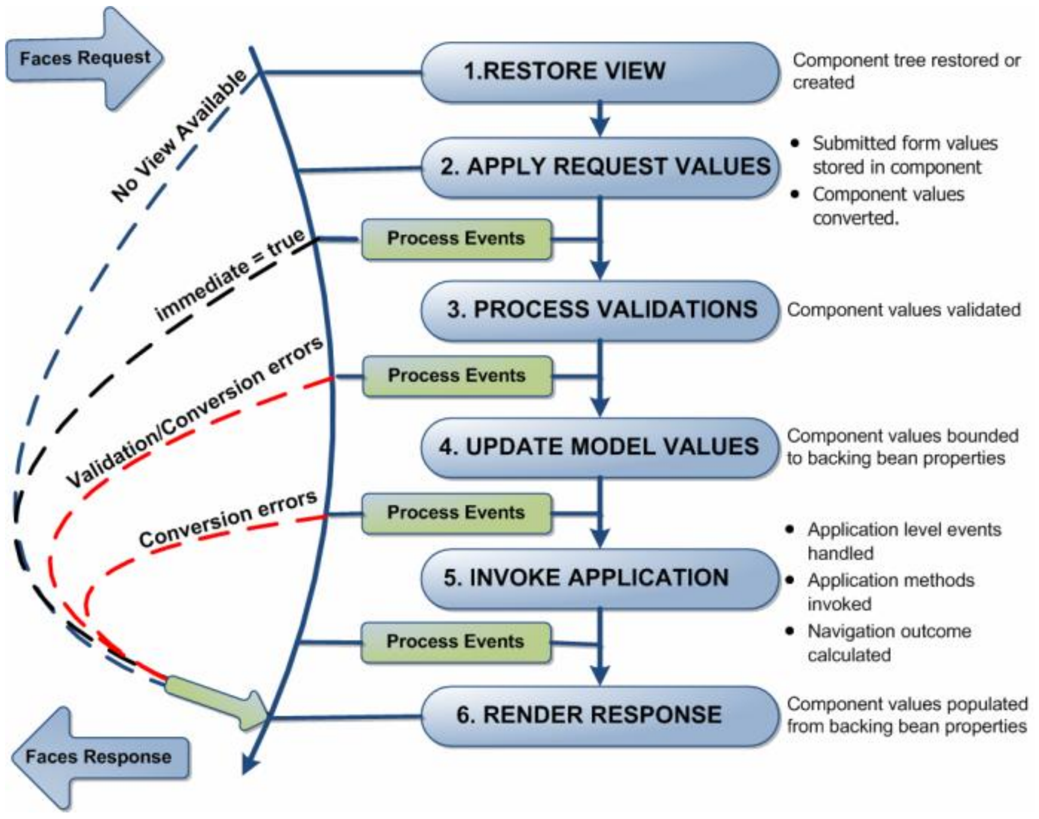
The JSF2 framework defines a lifecycle that executes in distinct phases, these phases are:

1. Restore View
2. Apply Request Values
3. Process Validations
4. Update Model Values
5. Invoke Application
6. Render Response

Lifecycle has two logical portions Execute and Render

– Supports AJAX partial processing and partial rendering

– You can specify components that JSF processes on the server or that JSF renders when an Ajax call returns



Faces Request
No View Available
1.RESTORE VIEW
Component tree restored or created
2. APPLY REQUEST VALUES
• Submitted form values stored in component
• Component values converted.
immediate = true
Process Events
3. PROCESS VALIDATIONS
Component values validated
Validation/Conversion errors
Process Events
4. UPDATE MODEL VALUES
Component values bounded to backing bean properties
Conversion errors
Process Events
• Application level events handled
• Application methods invoked
• Navigation outcome calculated
5. INVOKE APPLICATION
Process Events
6. RENDER RESPONSE
Component values populated from backing bean properties
Faces Response

Restore View

A request comes through the FacesServlet controller.
The controller examines the request and extracts the view ID, which is determined by the name of the xhtml page.

Apply request values

The purpose of the apply request values phase is for each component to retrieve its current state. The components must first be retrieved or created from the FacesContext object, followed by their values.

Process validations

In this phase, each component will have its values validated against the application's validation rules.

Update model value

In this phase JSF updates the actual values of the server-side model ,by updating the properties of your backing beans.
Invoke application   In this phase the JSF controller invokes the application to handle Form submissions.

Render respons
In this phase JSF displays the view with all of its components in their current state.

Now before understanding JSF life cycle, lets discuss something ie called PhaseListner(aka Servlet Listners)
That can interfere during various life cycle phases. We can used it to observe what is going to happens during various phases….

What is Phase Listners?

A PhaseListener is an Interface implemented on a java class that is registered with the JSF application
PhaseListeners provide hooks into the JSF lifecycle
Can be extremely helpful for application customization, optimization, and debugging
Can listen to all phases of the lifecycle or specific phases
The listener can be:
– Application wide with faces-config.xml
– Per page basis with <f:phaseListener>

Now to start let create another application, purpose of this application is to process job application by an applicant.

<u>Job-applicant.xhtml</u>

```
<h:body>
        <h:form>
                <h:panelGrid columns="2">
                        <h:outputLabel for="title" value="Title: " />
                        <h:selectOneRadio id="title" value="#{jobApplicant.title}">
                                <f:selectItem itemLabel="Dr." itemValue="1" />
                                <f:selectItem itemLabel="Ms." itemValue="2" />
                                <f:selectItem itemLabel="Mrs." itemValue="3" />
                                <f:selectItem itemLabel="Miss" itemValue="4" />
                                <f:selectItem itemLabel="Mr." itemValue="5" />
                        </h:selectOneRadio>
                        <h:outputLabel for="firstName" value="First Name: " />
                        <h:inputText id="firstName" value="#{jobApplicant.firstName}" />
                        <h:outputLabel for="lastName" value="Last Name: " />
                        <h:inputText id="lastName" value="#{jobApplicant.lastName}" />
                        <h:outputLabel for="country" value="Country: " />
                        <h:selectOneMenu id="country" value="#{jobApplicant.country}">
                                <f:selectItem itemLabel="-Select-" noSelectionOption="true" />
                                <f:selectItems value="#{countryList.countries}" />
                        </h:selectOneMenu>
                        <h:outputLabel for="salary" value="Salary: " />
                        <h:inputText id="salary" value="#{jobApplicant.salary}">
                                <f:convertNumber type="currency" integerOnly="true" />
                        </h:inputText>
                        <h:commandButton value="Submit Applicant" />
                </h:panelGrid>
        </h:form>
        <br />
        - Server -<br />
        Title: <h:outputText value="#{jobApplicant.title}" />
        <br />
        First Name: <h:outputText value="#{jobApplicant.firstName}" />
        <br />
        Last Name: <h:outputText value="#{jobApplicant.lastName}" />
        <br />
        Country: <h:outputText value="#{jobApplicant.country}" />
        <br />
        Salary: <h:outputText value="#{jobApplicant.salary}" />
        <br />
```

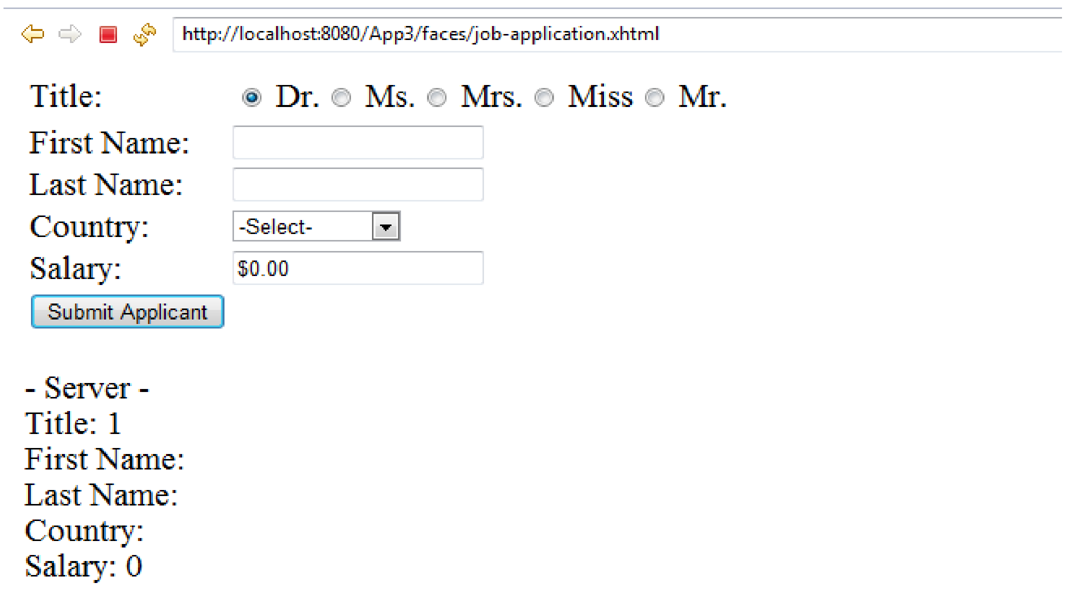
model

```
package com.beans;

import java.io.Serializable;

import javax.faces.bean.ManagedBean;
import javax.faces.bean.ViewScoped;

@ManagedBean(name="jobApplicant")
@ViewScoped
public class JobApplicant implements Serializable {

        private static final long serialVersionUID = 1L;
        private String firstName;
        private String lastName;
        private Integer title;
        private String country;
        private int salary;

        public JobApplicant(){}
        //getter and setter
}
```

for countries

```
package com.beans;

import javax.faces.bean.ApplicationScoped;
import javax.faces.bean.ManagedBean;

@ManagedBean(name="countryList")
@ApplicationScoped
public class CountryList {

        private String[] countries = { "Canada", "United States" };

        public String[] getCountries() {
                return countries;
        }

        public void setCountries(String[] countryList) {
                this.countries = countryList;
        }
}
```

Now run the application and play with it……..



http://localhost:8080/App3/faces/job-application.xhtml

Title: [✓ Dr.] Ms. Mrs. Miss Mr.

First Name:

Last Name:

Country: -Select-

Salary: $0.00

Submit Applicant

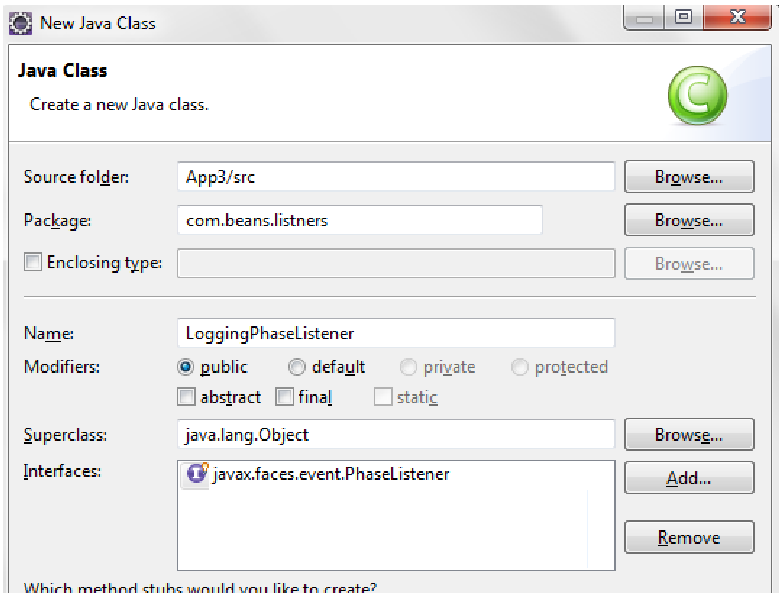
- Server -
Title: 1
First Name:
Last Name:
Country:
Salary: 0

Now our motive is to understand various life cycle phases….
So we are going to add PhaseListener to the jobApplication project

This PhaseListener will simply log the Phase ID to the Tomcat console log, before and after phases execute

Creating phase listner

step 1:Add and listner as shown


New Java Class

Java Class

Create a new Java class.

Source folder: App3/src

Package: com.beans.listners

Enclosing type: [ ]

Name: LoggingPhaseListener

Modifiers: [✓ public] default private protected

[ ] abstract [ ] final [ ] static

Superclass: java.lang.Object

Interfaces: javax.faces.event.PhaseListener

```
public class LoggingPhaseListener implements PhaseListener {

        private static final long serialVersionUID = 1L;

        ………..

}
```

step 2: override methods

```
   @Override
        public void afterPhase(PhaseEvent event) {
                System.out.println("AFTER PHASE: " + event.getPhaseId().toString());
        }

        @Override
        public void beforePhase(PhaseEvent event) {
                System.out.println("BEFORE PHASE: " + event.getPhaseId().toString());
        }

        @Override
        public PhaseId getPhaseId() {
                return PhaseId.ANY_PHASE;
        }
```

step 3: register listner

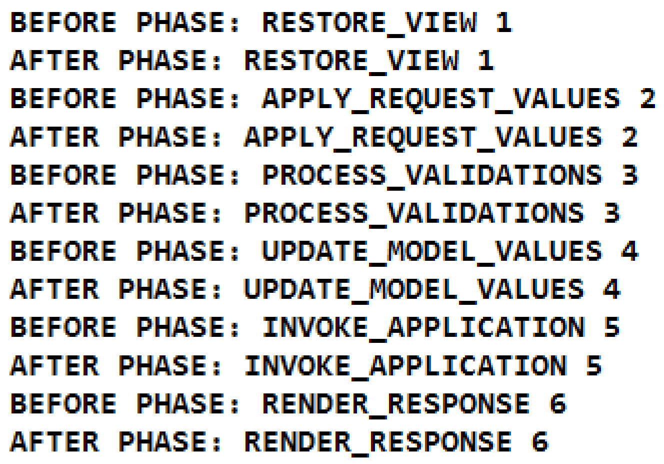
LoggingPhaseListener class to be registered as a PhaseListener when the JSF framework initializes

add following in jsf-config.xml

```
        <lifecycle>
        <phase-listener>
```

```
                        com.beans.listners.LoggingPhaseListener
            </phase-listener>
            </lifecycle>
```

step 4: ovserve the log



```
BEFORE PHASE: RESTORE_VIEW 1
AFTER PHASE: RESTORE_VIEW 1
BEFORE PHASE: APPLY_REQUEST_VALUES 2
AFTER PHASE: APPLY_REQUEST_VALUES 2
BEFORE PHASE: PROCESS_VALIDATIONS 3
AFTER PHASE: PROCESS_VALIDATIONS 3
BEFORE PHASE: UPDATE_MODEL_VALUES 4
AFTER PHASE: UPDATE_MODEL_VALUES 4
BEFORE PHASE: INVOKE_APPLICATION 5
AFTER PHASE: INVOKE_APPLICATION 5
BEFORE PHASE: RENDER_RESPONSE 6
AFTER PHASE: RENDER_RESPONSE 6
```

Next we are going to learn something about **JSF Validators**

After understanding validator we will come back on JSF life cycle…Don’t worry..

6. JSF Validators

JSF utilizes validators to ensure user input matches criteria specified by the developer
This is achieved through:

– 6 standard built in JSF 2.0 validators
– required attribute on components
– Interdependent validation in the backing bean
– Custom validators (in a class or bean method)

JSF provides 6 built in validators:

- validateDoubleRange
  - min/max range for a Double
- validateLongRange
  - min/max range for a Long
- ValidateLength
  - min/max input size
- ValidateBean
  - Used in conjunction with Bean Validation API
  - check preset business logic restraints
- validateRegex
  - compare to a regular expression String
- validateRequired
  - same as required attribute

Attribute "required"

JSF provides an attribute to ensure an input field is populated by the user
Use required="true/false"
For example:
<h:inputText value="#{modelBean.firstName}“ required="true"/>

How Validation is Performed

As part of the JSF Lifecycle a phase called PROCESS_VALIDATIONS is used
The lifecycle will be covered later
If validation fails JSF will generate an error message that can be displayed on the page
Validation error messages are displayed using <h:message>, for example:
<h:inputText id="firstName" value="#{modelBean.firstName}"
required="true"/>
<h:message for="firstName"/>

Custom Validator Class

Creating a custom validator is useful if you need to check input in a way not provided by the standard validators
For example:

Country validation
Username uniqueness validation
Email format validation

We will step through the process of creating a custom validator latter let first appliy build in validator in our existing application ….

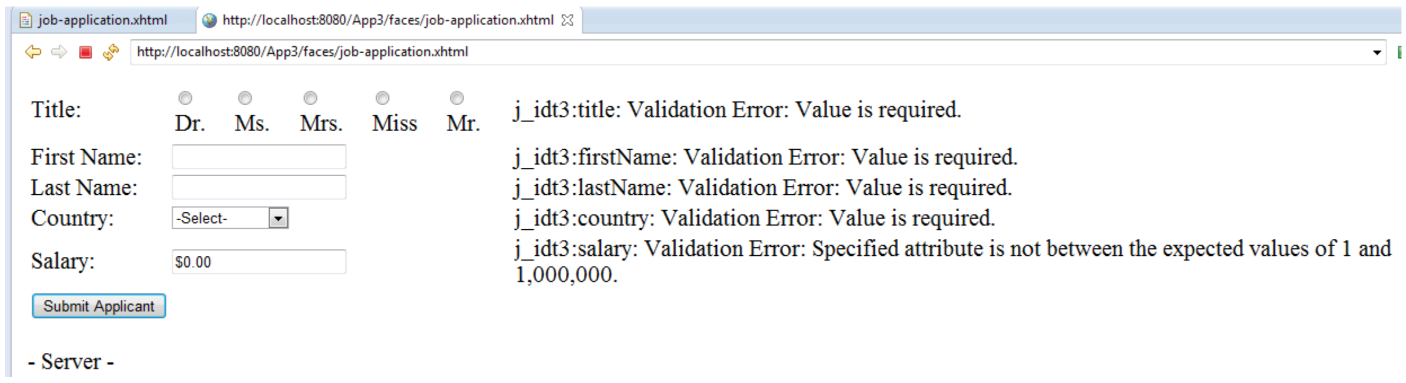
Now modify earlier application xhtml page to incoporate JSF validation ….

final job applicant

```
<h:form>
                    <h:messages globalOnly="true" />
                    <h:panelGrid columns="3">
                              <h:outputLabel for="title" value="Title: " />
                              <h:selectOneRadio id="title" required="true"
                                        value="#{jobApplicant.title}">
                                        <f:selectItem itemLabel="Dr." itemValue="1" />
                                        <f:selectItem itemLabel="Ms." itemValue="2" />
                                        <f:selectItem itemLabel="Mrs." itemValue="3" />
                                        <f:selectItem itemLabel="Miss" itemValue="4" />
                                        <f:selectItem itemLabel="Mr." itemValue="5" />
                              </h:selectOneRadio>
                              <h:message for="title" />
                              <h:outputLabel for="firstName" value="First Name: " />
                              <h:inputText id="firstName" value="#{jobApplicant.firstName}"
                                        required="true" />
                              <h:message for="firstName" />
                              <h:outputLabel for="lastName" value="Last Name: " />
                              <h:inputText id="lastName" value="#{jobApplicant.lastName}"
                                        required="true" />
                              <h:message for="lastName" />

                              <h:outputLabel for="country" value="Country: " />
                              <h:selectOneMenu id="country" value="#{jobApplicant.country}"
                                        required="true">
                                        <f:selectItem itemLabel="-Select-" noSelectionOption="true" />
                                        <f:selectItems value="#{countryList.countries}" />
                              </h:selectOneMenu>
                              <h:message for="country" />
                              <h:outputLabel for="salary" value="Salary: " />
                              <h:inputText id="salary" value="#{jobApplicant.salary}"
                                        required="true">
                                        <f:convertNumber type="currency" integerOnly="true" />
                                        <f:validateLongRange minimum="1" maximum="1000000" />
                              </h:inputText>
                              <h:message for="salary" />
                              <h:commandButton actionListener="#{jobApplicant.submit}"
                                        value="Submit Applicant" />
                    </h:panelGrid>
          </h:form>
          <br />
  - Server -<br />
  Title: <h:outputText value="#{jobApplicant.title}" />
          <br />
  First Name: <h:outputText value="#{jobApplicant.firstName}" />
          <br />
  Last Name: <h:outputText value="#{jobApplicant.lastName}" />
          <br />
  Country: <h:outputText value="#{jobApplicant.country}" />
          <br />
  Salary: <h:outputText value="#{jobApplicant.salary}" />
          <br />
```

Focus on highlighted text , it is not difficult to guss the purpose of the tags incorporated.

Run it and Observe following output…


job-application.xhtml
http://localhost:8080/App3/faces/job-application.xhtml
http://localhost:8080/App3/faces/job-application.xhtml

Title: ○ Dr. ○ Ms. ○ Mrs. ○ Miss ○ Mr.

First Name:

Last Name:

Country: -Select-

Salary: $0.00

Submit Applicant

- Server -

```
j_idt3:title: Validation Error: Value is required.
j_idt3:firstName: Validation Error: Value is required.
j_idt3:lastName: Validation Error: Value is required.
j_idt3:country: Validation Error: Value is required.
j_idt3:salary: Validation Error: Specified attribute is not between the expected values of 1 and
1,000,000.
```

Now you observe that error message looks really bad, how to customized it(hold down I will be discuss it with internationalization)

Interdependent Field Validation – ActionListener

Now lets say what my bussiness logic want that I should check firstname and last name from database and if they already exist then I should not allow user to be created ..what to do?

Assume that firstName: foo
lastName:bar
is already exist, so I should not allow to enter that user again……..

Steps
Step:1
Paste following code in class JobApplicant

```java
public void submit(ActionEvent ae) {
        if (firstName.equals("foo") && lastName.equals("bar")) {
            String msg = "foo bar already works for us";
            FacesMessage facesMessage = new FacesMessage(msg);
            FacesContext facesContext = FacesContext.getCurrentInstance();
            String clientId = null; // this is a global message
            facesContext.addMessage(clientId, facesMessage);
        }
    }
```

Step 2;
Replace earlier command button with this……

```xml
<h:commandButton actionListener="#{jobApplicant.submit}"
                 value="Submit Applicant" />
```

The new **actionListener** attribute is bound to the jobApplicant.submit() method.

Step 3;
Add following after<h:form> tags
`<h:messages globalOnly="true" />`

Now Validation should fail with the global error message “foo bar already works for us”

Custom Validator Exercise

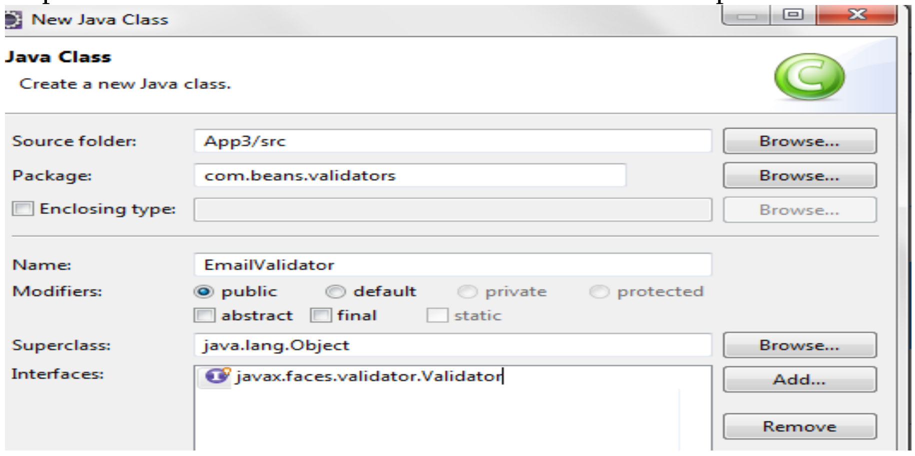
Now lets add an email field and we only want to allow valid email into it.

Step 1;
Add after last name field:

```
<h:outputLabel for="email" value="email:" />
<h:inputText id="email" value="#{jobApplicant.email}"
        validator="emailValidator" required="true" />
<h:message for="email" />
```

Step 2: add email field to our bean with getter and setter…….

Step 3: Create an validator class EmailValidator that implements Validator inteface


New Java Class

Java Class

Create a new Java class.

Source folder: App3/src

Package: com.beans.validators

Enclosing type:

Name: EmailValidator

Modifiers: [✓ public] default private protected

abstract final static

Superclass: java.lang.Object

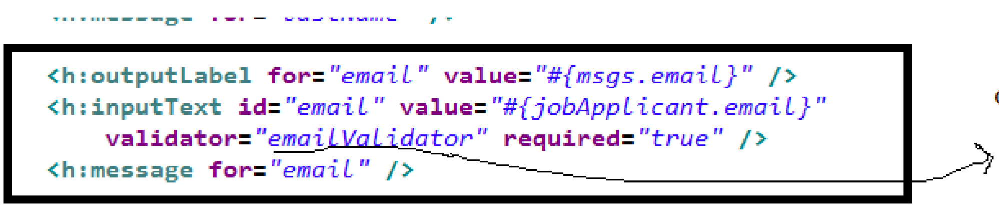
Interfaces: javax.faces.validator.Validator

Step 4:
Add @FacesValidator("emailValidator") over it, so that it get register with framework.

Step 5:
Implement validate() method
As

```
@Override
        public void validate(FacesContext arg0, UIComponent arg1, Object arg2)
                        throws ValidatorException {
                String inputEmail = (String) arg2;
                // Set the email pattern string. (?i) is a flag for case-insensitive.
                Pattern p = Pattern.compile("(?i)\\b[A-Z0-9._%-]+@[A-Z0-9.-]+\\.[A-Z]{2,4}\\b");
                // Match the given string with the pattern
                Matcher m = p.matcher(inputEmail);
                // Check whether match is found
                boolean matchFound = m.matches();
                if (!matchFound) {
                        String invalidEmailMsg = "Invalid Email";
                        FacesMessage message = new FacesMessage(
                                        FacesMessage.SEVERITY_ERROR, invalidEmailMsg,
                                        invalidEmailMsg);
                        throw new ValidatorException(message);
                }
        }
```

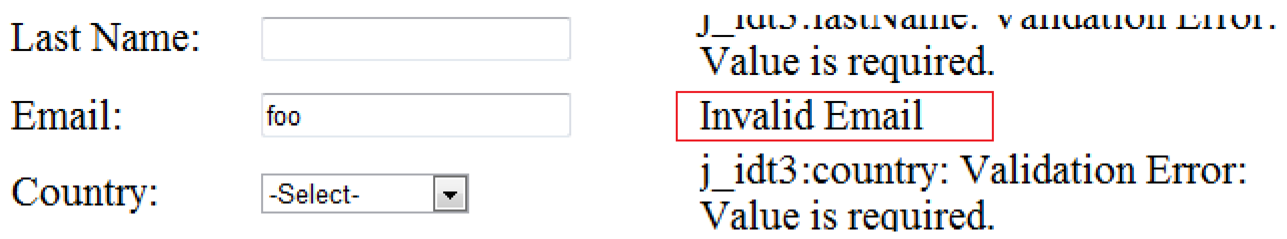
Now you must have understand that we have to refer this validator in <h:inputText>



```
<h:outputLabel for="email" value="#{msgs.email}" />
<h:inputText id="email" value="#{jobApplicant.email}"
    validator="emailValidator" required="true" />
<h:message for="email" />
```

custom validators

Now ovserve the output if you provide invalid email id



Last Name:		j_idt3:lastName: Validation Error: Value is required.
Email:	foo	Invalid Email
Country:	-Select-	j_idt3:country: Validation Error: Value is required.

7. JSF Converters

JSF uses converters to modify input and format output

- Conversion options:
  - – Implicit or Explicit conversion of datatypes
  - – Standard convertDateTime and convertNumber tags
  - – Create Custom Converters

JSF implicitly converts bound values to the proper datatype,for example:
On the page:
<h:inputText value="#{modelBean.age}"/>
– In the backing bean:
private int age;
public int getAge() { return age; }
public void setAge(int age) { this.age = age; }

- Instead of requiring a String datatype JSF can handle the int age automatically

Standard Converter Tag Examples

```
f:convertDateTime usage:
    <h:inputText value="#{modelBean.dateOfBirth}">
    <f:convertDateTime type="date"
    timeZone="#{zoneUtil.currentTimeZone}"
    pattern="MM/dd/yy"/>
</h:inputText>

f:convertNumber usage:
    <h:inputText value="#{modelBean.accountBalance}">
    <f:convertNumber type="currency" groupingUsed="true"
    minIntegerDigits="2" maxIntegerDigits="2"
    currencyCode="$"/>
</h:inputText>
```

<u>Custom Converters</u>

Creating a custom converter is useful if you need to format input or output to match your business logic
– For example:

Social Security number converter
Phone number converter
Credit card converter

Next i step through the process of creating a custom converter

The goal of this example is to Capitalize our firstName and lastName input using a custom Converter

Step 1:
Create an Converter class

```
public class WordCapatilizationConverter implements Converter{

	@Override
	public Object getAsObject(FacesContext arg0, UIComponent arg1, String arg2) {
		// TODO Auto-generated method stub
		return null;
	}

	@Override
	public String getAsString(FacesContext arg0, UIComponent arg1, Object arg2) {
		// TODO Auto-generated method stub
		return null;
	}

}
```

Step 2: Annotate with

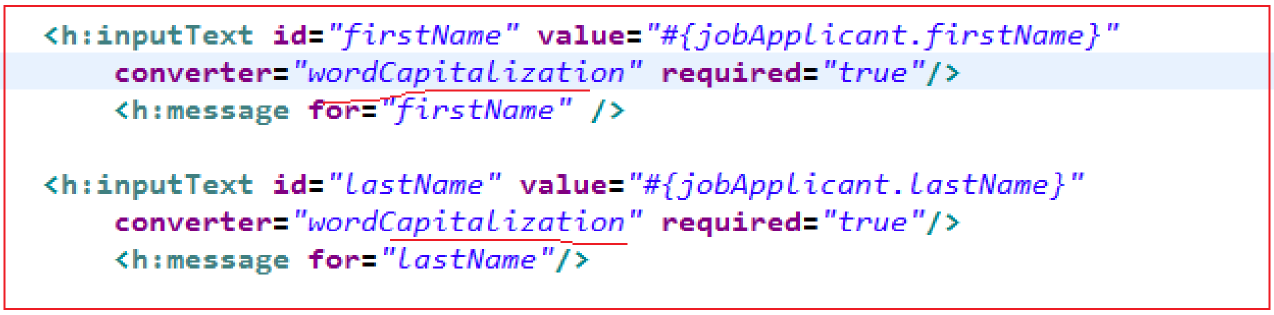
```
@FacesConverter("wordCapitalization")
```

Step 3:
Override methods and provide logic to word capatilization

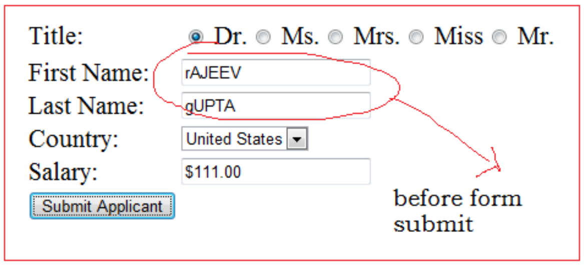
```
	@Override
	public Object getAsObject(FacesContext arg0, UIComponent arg1, String arg2) {
		return fixCapitalization(arg2);
	}

	@Override
	public String getAsString(FacesContext arg0, UIComponent arg1, Object arg2) {
		return fixCapitalization((String) arg2);
	}

	private String fixCapitalization(String inString) {
		if (inString == null) {
			return "";
		}
		StringBuffer str = new StringBuffer(inString.trim().toLowerCase());
		if (str.length() == 0) {
			return str.toString();
		}
		Character nextChar;
		int i = 0;
		nextChar = new Character(str.charAt(i));
		while (i < str.length()) {
			str.setCharAt(i++, Character.toUpperCase(nextChar.charValue()));
			if (i == str.length()) {
				return str.toString();
			}
			// Look for whitespace
			nextChar = new Character(str.charAt(i));
			while (i < str.length() - 2
					&& !Character.isWhitespace(nextChar.charValue())) {
				nextChar = new Character(str.charAt(++i));
			}
			if (!Character.isWhitespace(nextChar.charValue())) {
				// If not whitespace, we must be at end of string
				return str.toString();
			}
			// Remove all but first whitespace
			nextChar = new Character(str.charAt(++i));
			while (i < str.length()
					&& Character.isWhitespace(nextChar.charValue())) {
				str.deleteCharAt(i);
				nextChar = new Character(str.charAt(i));
			}
		}
		return str.toString();
	}
```

Step 4: Apply conversion to view

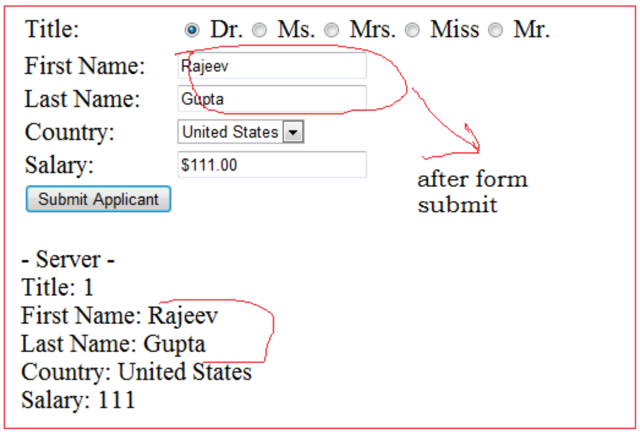
```
<h:inputText id="firstName" value="#{jobApplicant.firstName}"
            converter="wordCapitalization" required="true"/>
<h:message for="firstName" />

<h:inputText id="lastName" value="#{jobApplicant.lastName}"
            converter="wordCapitalization" required="true"/>
<h:message for="lastName"/>
```



```
<h:inputText id="firstName" value="#{jobApplicant.firstName}"
    converter="wordCapitalization" required="true"/>
    <h:message for="firstName" />

<h:inputText id="lastName" value="#{jobApplicant.lastName}"
    converter="wordCapitalization" required="true"/>
    <h:message for="lastName"/>
```



Title:	[● Dr.] ○ Ms. ○ Mrs. ○ Miss ○ Mr.
First Name:	rAJEEV
Last Name:	gUPTA
Country:	United States
Salary:	$111.00



Title:	[● Dr.] ○ Ms. ○ Mrs. ○ Miss ○ Mr.
First Name:	Rajeev
Last Name:	Gupta
Country:	United States
Salary:	$111.00

```
- Server -
Title: 1
First Name: Rajeev
Last Name: Gupta
Country: United States
Salary: 111
```

8. Internationalization

Internationalization

JSF has full support for Java il8n
- You can use internationalization in pages and beans
- Configure your supported languages in faces-config.xml
- JSF provides the ability to dynamically switch locales
- Can also override default JSF message and error text

faces-config.xml Supported Locales

The <locale-config> element is used in facesconfig.xml

Specify a default locale and any supported locales
– Use the lower-case, two-letter codes as defined by ISO-639

```
<application>
    <locale-config>
    <default-locale>en</default-locale>
    <supported-locale>de</supported-locale>
    <supported-locale>it</supported-locale>
    </locale-config>
</application>
```

Note: The above code would support English, German, and Italian

faces-config.xml Resource Bundles

Strings are added to properties files and registered with JSF using the <resource-bundle> element in faces-config.xml

Specify a base-name used by each properties file:

```
<application>
    <resource-bundle>
            <base-name>
            com.messages
            </base-name>
    <var>msgs</var>
    </resource-bundle>
</application>
```

We create a "messages_xy.properties" file for each supported locale
– Where "xy" is our two letter language identifier
– The files are placed in the specified package

Language Files

In each "messages_xy.properties" file we would specify:

– Key used in our page/bean
– Internationalized value for the key

For example the content of messages_en.properties:

```
firstName=First Name
lastName=Last Name
submit=Submit
```

And the content of messages_de.properties:

```
firstName=Vorname
lastName=Nachname
submit=Reichen Sie ein
```

Using Internationalization in Pages

In our page we can access these properties files using the EL (Expression Language)
The EL uses the <var> element specified in the <resource-bundle> to retrieve key values:

```
<h:outputText value="#{msgs['firstName']}" />
```

Or

```
<h:outputText value="#{msgs.firstName}" />
```

Would display:

– "First Name" in English
"Vorname" in German

Lets start modifiying existing application to support internationlization

Step 1: Create Resource Bundle

Create a file Msgs_en.properties in a new com.resources package and paste the following into it:

```
title=Title
firstName=First Name
lastName=Last Name
email=Email
country=Country
salary=Salary
submitApplicant=Submit Applicant
```

Let create another resource bunld for german language

```
messages_de.properties:
      firstName=Vorname
      lastName=Nachname
      submit=Reichen Sie ein
```

Step 2:

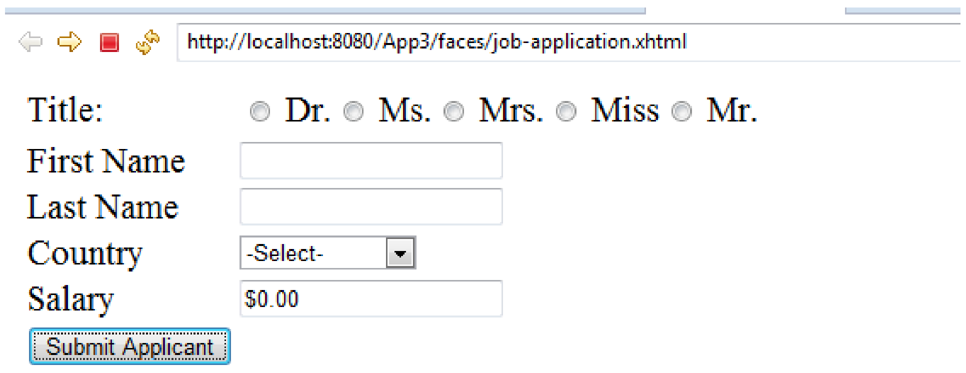
In the faces-config.xml file, add the resource bundle to the application node:

```
<application>
        <resource-bundle>
                <base-name>com.resources.Msgs</base-name>
                <var>msgs</var>
        </resource-bundle>
</application>
```

Step 3: change value binding

```
<h:outputLabel for="title" value="#{msgs.title}" />
<h:outputLabel for="firstName" value="#{msgs.firstName}" />
<h:outputLabel for="lastName" value="#{msgs.lastName}" />
<h:outputLabel for="email" value="#{msgs.email}" />
<h:outputLabel for="country" value="#{msgs.country}" />
<h:outputLabel for="salary" value="#{msgs.salary}" />
<h:commandButton value="#{msgs.submitApplicant}" .../>
```

Now run the application , you find label messages are coming from resource bundle



Now pending question how to customized error message in jsf 2, We have power to override default messages usign

JSF Message Keys

JSF-override_xy.properties can reference existing keys

For example the JSF key for the default required message:
javax.faces.component.UIInput.REQUIRED

In the properties file override the value:

```
javax.faces.component.UIInput.REQUIRED=Missing a value.
```

Now changes the validation message for required="true“ to "Missing a value."

Override Standard JSF Messages

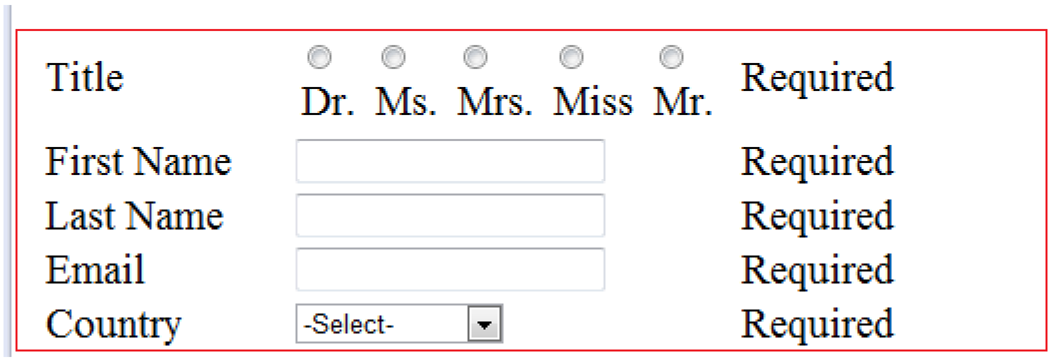
In order to override standard JSF messages create a file in the com.resources package named JSF-override_en.properties
Paste the following key override into it:

```
javax.faces.component.UIInput.REQUIRED=Required
```

Open the faces-config.xml file and add the following entry to the <application> section:

```
<application>
    <message-bundle>
            com.resources.JSF-override
    </message-bundle>
    ...
</application>
```

Now run and observe,that it print customized error messages



Title	○ Dr. ○ Ms. ○ Mrs. ○ Miss ○ Mr.	Required
First Name		Required
Last Name		Required
Email		Required
Country	-Select-	Required