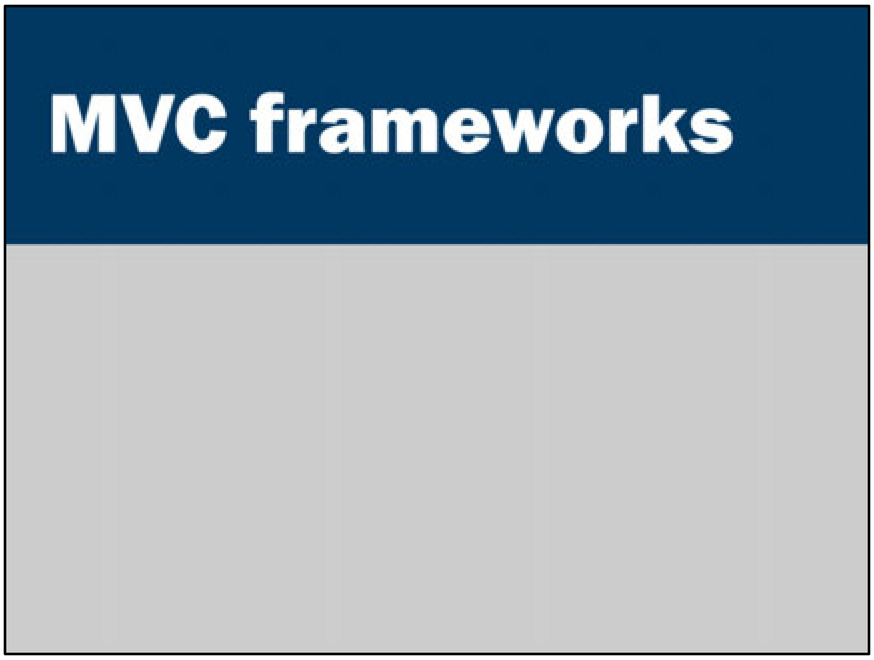
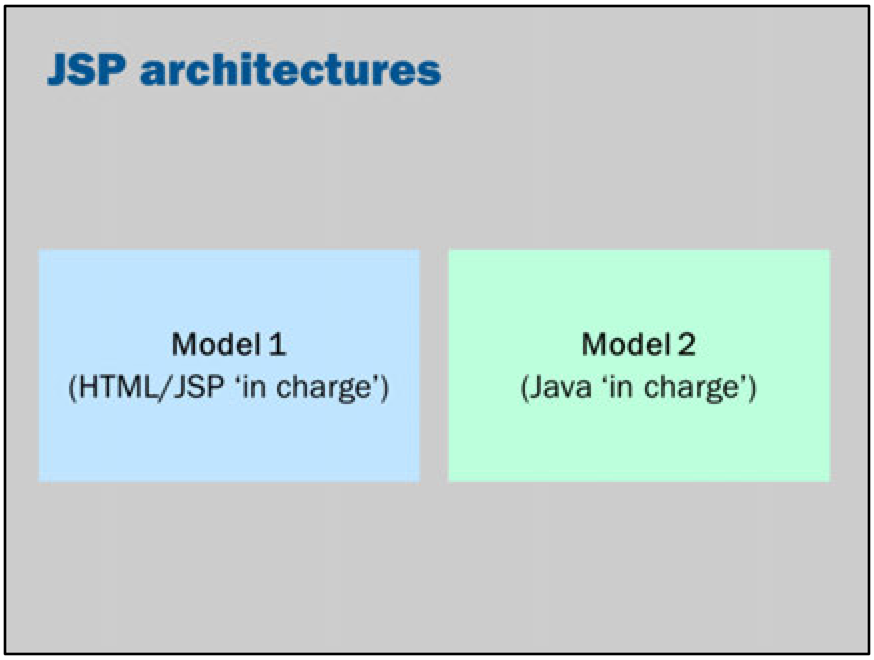


MVC frameworks

More reading:
A series about real world projects that use JavaServer Faces:
http://www.jsfcentral.com/trenches

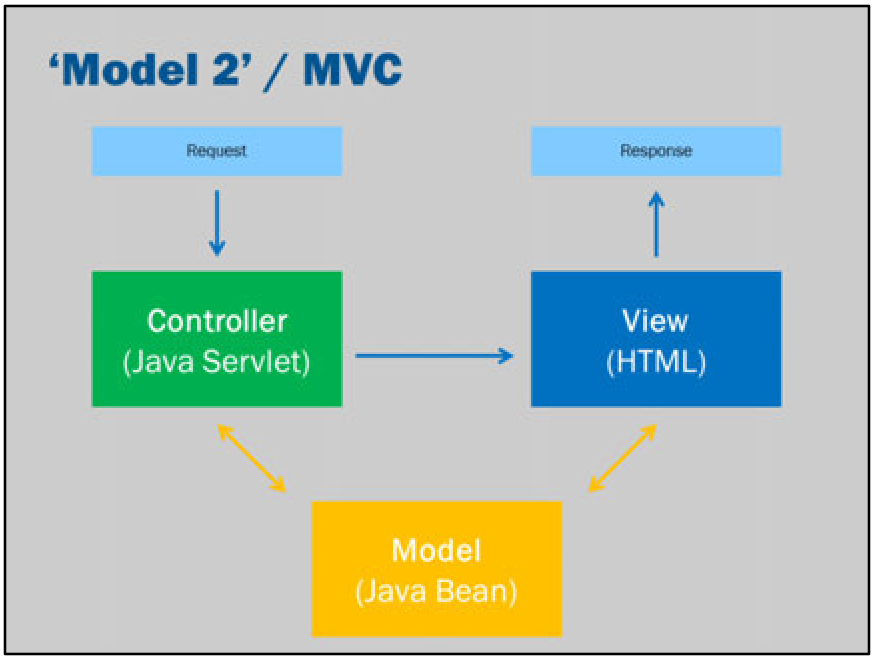


JSP architectures

Model 1
(HTML/JSP ‘in charge’)

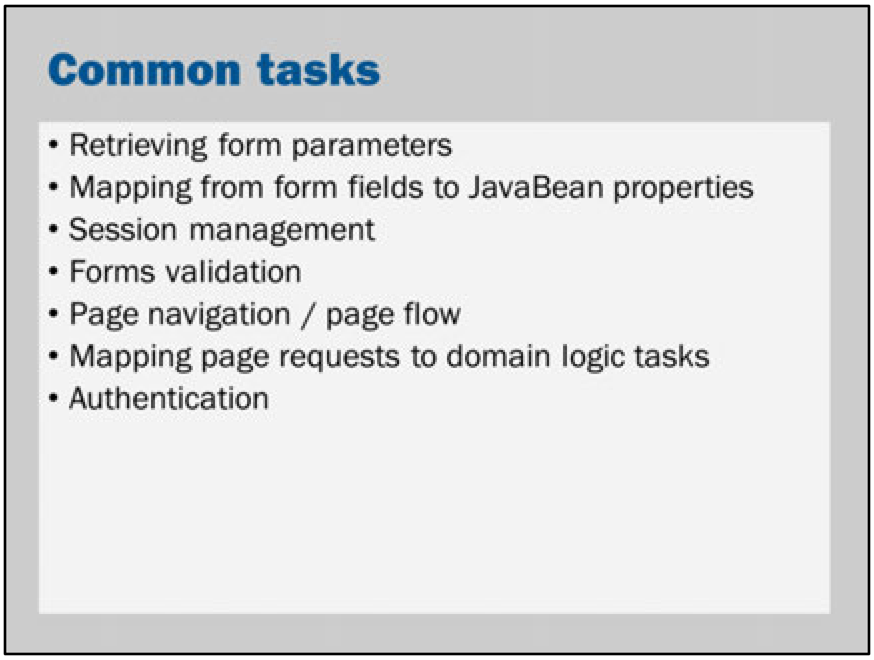
Model 2
(Java ‘in charge’)

This is just a revision slide.



‘Model 2’ / MVC

Another revision slide.



Common tasks

- Retrieving form parameters
- Mapping from form fields to JavaBean properties
- Session management
- Forms validation
- Page navigation / page flow
- Mapping page requests to domain logic tasks
- Authentication

What are some common tasks/problems that need to be solved in most web applications?

- **Retrieving form parameters**
  i.e., in the Servlet/JSP, using code such as: request.getParameter("value")
- **Mapping form fields to class fields**
  i.e., model.setValue(request.getParameter("value"));
- **Session management**
  i.e.,
  ```
  Model model = (Model)request.getSession().getAttribute("model");
  if (model == null) {
    model = new Model();
    request.getSession().setAttribute(model);
  }
  ```
- **Form validation**
  i.e.,
  ```
  try {
    int value = Integer.parseInt(request.getValue("value"));
  } catch (NumberFormatException nfe) {
    showErrorMessage("Please enter a valid value");
  ```

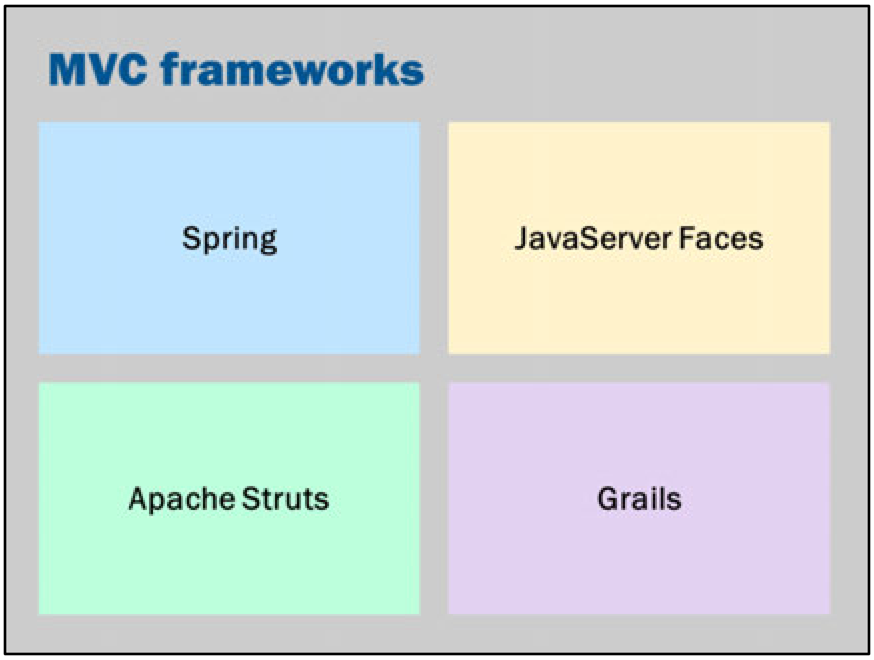
}

- **Page navigation / page flow**
  i.e., what should be the “??????” in <a href=“??????”>Next Page</a>
- **Mapping page requests to a domain logic task/action**
  i.e., if (request.getParameter(“submit-button”) != null) {
    model.doAction();
  }
- **Authentication**
  i.e.,
  if (request.getSession().getAttribute(“user”) == null) {
    // show logon screen
  } else {
    // check authentication
  }

This brings us the need for frameworks.
Frameworks provide a structure for software development projects.
Frameworks provide common ‘plumbing’ or ‘boilerplate’ tasks.

The model view controller architecture/pattern is very popular within web development.
So, the need for MVC and a desire to reduce duplication brings us to use MVC frameworks.

Once such MVC framework is JavaServer Faces, and we will use this in the lecture.


MVC frameworks

Spring

JavaServer Faces

Apache Struts

Grails

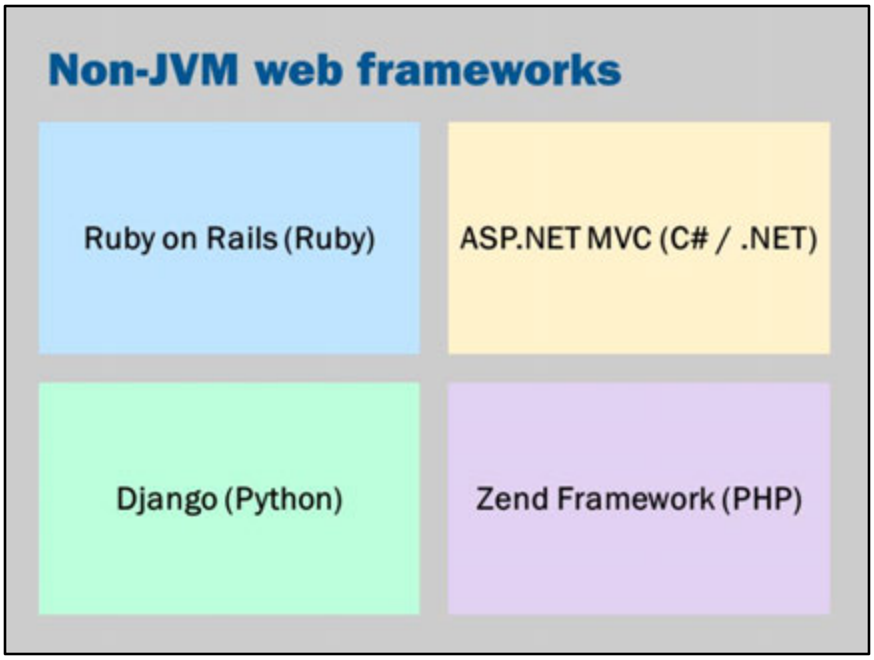
JavaServer Faces isn't the only framework to choose from…

http://en.wikipedia.org/wiki/Comparison_of_web_application_frameworks#Java

Spring and Struts are older frameworks.
They're very popular.
JSF is also popular and growing in popularity.
JSF is recommended by the Java EE specifications.
I expect that with this "official endorsement" its popularity will continue to grow.

Grails uses a programming language called Groovy. However, it runs on the Java virtual machine.

We will be using JSF in this subject because of its "official endorsement".
Once you've learnt JSF, you will be able to understand concepts in other MVC frameworks.



Non-JVM web frameworks

Ruby on Rails (Ruby)

ASP.NET MVC (C# / .NET)

Django (Python)

Zend Framework (PHP)

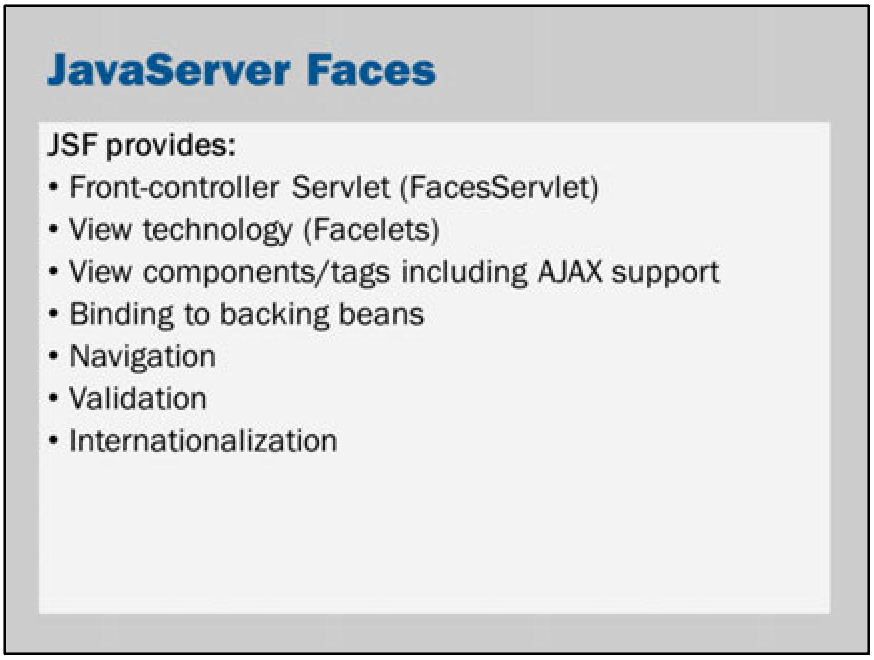
MVC/web frameworks aren't unique to Java either…

"Ruby on Rails" for the Ruby programming language is the 'classic' MVC framework.
Created in 2004, It started the trend of MVC frameworks in web development.

On .NET, there is ASP.NET MVC.

Virtually every mainstream programming language has its own web frameworks.
Once you understand one MVC framework, you'll find it very easy to switch to other frameworks.
So, it doesn't really matter that much which one you learn first.

Historically, MVC comes from desktop software development.
It was first invented in the 1970s at Xerox Parc (the 'home' of desktop computers).



JavaServer Faces

JSF provides:

- Front-controller Servlet (FacesServlet)
- View technology (Facelets)
- View components/tags including AJAX support
- Binding to backing beans
- Navigation
- Validation
- Internationalization

JavaServer Faces is created from a number of separate technologies.

It provides its own Servlet, javax.faces.webapp.FacesServlet.
This Servlet does all the hard work of handling requests.

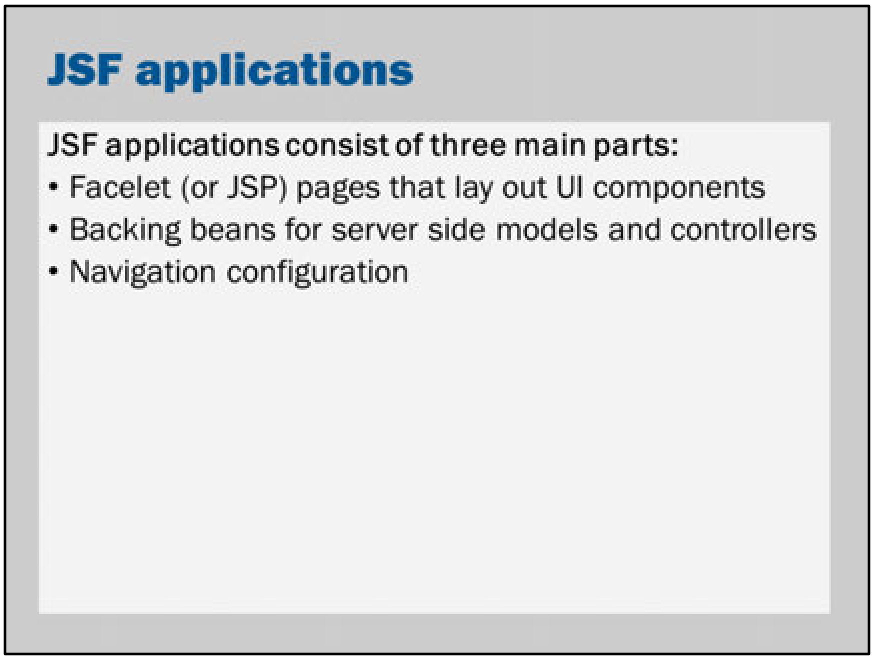
It uses a view technology called Facelets.
Facelets *replaces* JSP.
Facelets is an XML-based file format (XHTML, actually).
The use of XML helps ensure the output is always valid.
This helps address the problem of JSP being “difficult to check”.

JSP - Pages - is a HTML templating langage.
It generates HTML directly.
JSF - Faces - is, in contrast, a view definition language.
The elements of a Facelets file represent user interface components.
The components generate their own HTML.

JSF provides a number of features “behind the scenes”.
These may be thought of as capabilities of the FacesServlet.
They’re handled for you automatically:

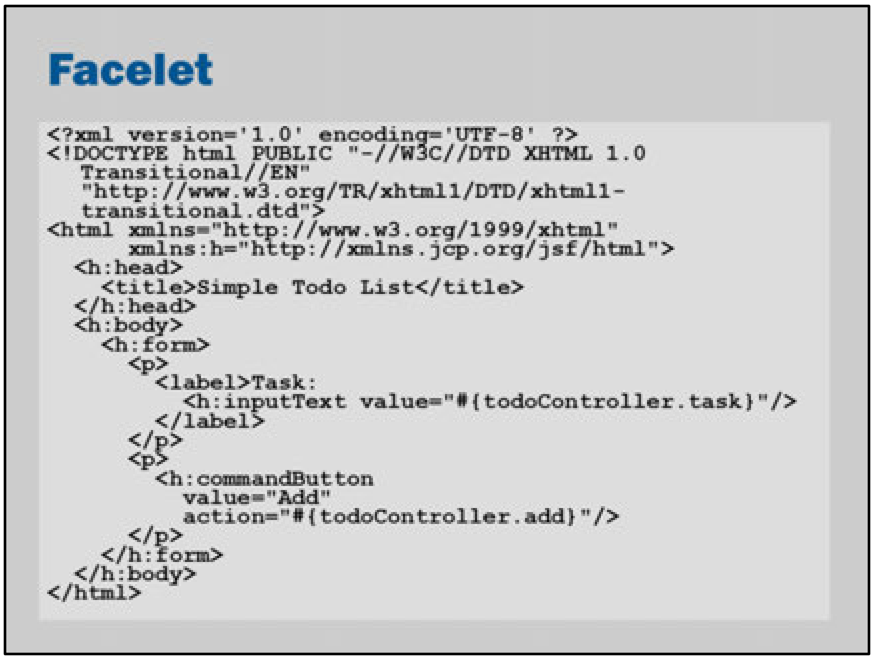
- Binding to backing beans (linking user interface components to your own Java code)
- Navigation (deciding which page to go to when an action is performed)
- Validation (checking to make sure user input is correct)
- Internationalization (creating websites in multiple languages)

JavaServer Faces


JSF applications

JSF applications consist of three main parts:
- Facelet (or JSP) pages that lay out UI components
- Backing beans for server side models and controllers
- Navigation configuration

When you write your own JSF application, you will create:
- One or more view files written using Facelets (i.e., XHTML).
  These represent the user interface.
- One or more backing beans.
  Backing beans are ordinary Java Beans that are connected to your views.
  The backing beans act as the controllers for your application
- An optional configuration file that tells JSF which pages should be shown when users click on links.
- Optionally, custom converters, event listeners and user interface components.
  These extend the basic functionality of JSF.
  However, they are more complex to build and would only appear in more advanced projects.



Facelet

```
<?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://xmlns.jcp.org/jsf/html">
  <h:head>
    <title>Simple Todo List</title>
  </h:head>
  <h:body>
    <h:form>
      <p>
        <label>Task:
          <h:inputText value="#{todoController.task}"/>
        </label>
      </p>
      <p>
        <h:commandButton
          value="Add"
          action="#{todoController.add}"/>
      </p>
    </h:form>
  </h:body>
</html>
```

Here is a simple Facelet.
Note that it uses XML/XHTML syntax instead of JavaServer Pages.
This is a strictly defined syntax it is easily checked/verified.

Things to note:

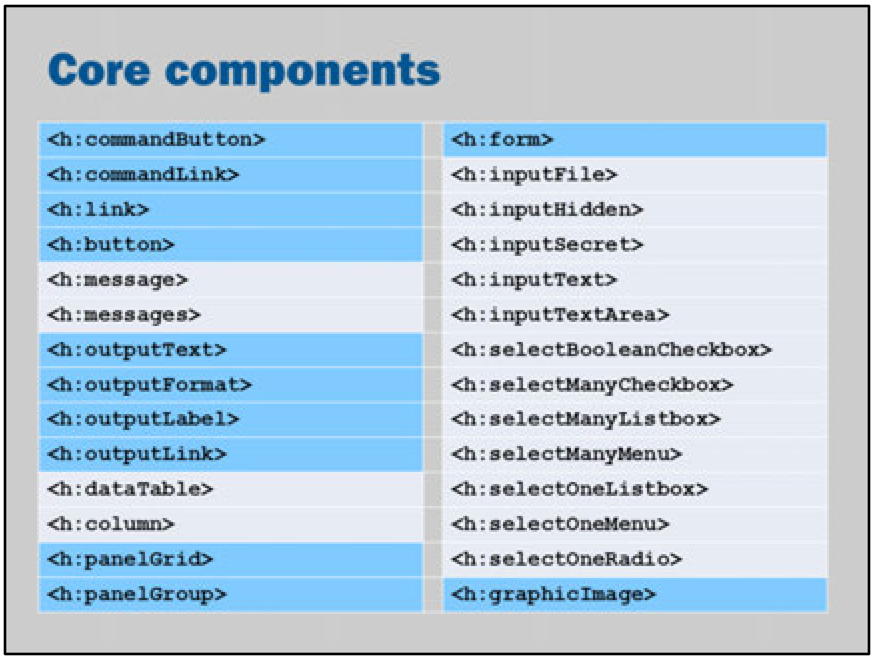
Some tags have prefixes: instead of <body> and <form>, we have <h:body> and <h:form>.
This "h:" prefix refers to an XML namespace.
The XML namespace is declared in the <html> element:
    xmlns:h="http://xmlns.jcp.org/jsf/html"
XMLNS stands for XML Name Space.
The use of namespaces tells Facelets that the <h:body> tag isn't a standard HTML tag.
It tells Facelets that this tag refers to a Facelets user interface component.
In other words, <h:body> is a Facelets user interface component.

Some of the components on this page are interesting:
<h:inputText> is a UI component that corresponds to <input type="text">
<h:commandButton> is a UI component that corresponds to <input type="submit">

or <button>

The <h:form>, <h:body> and <h:head> tags are comparatively simple tags.
They just generate the corresponding HTML tags.
However, they are important because JSF needs to add its own extra information into these tags.
For example, a component in the body may need stylesheets to be added to the <head>.

Finally, notice that we are using Expression Language.
However, the expressions are written using # instead of $.



Core components

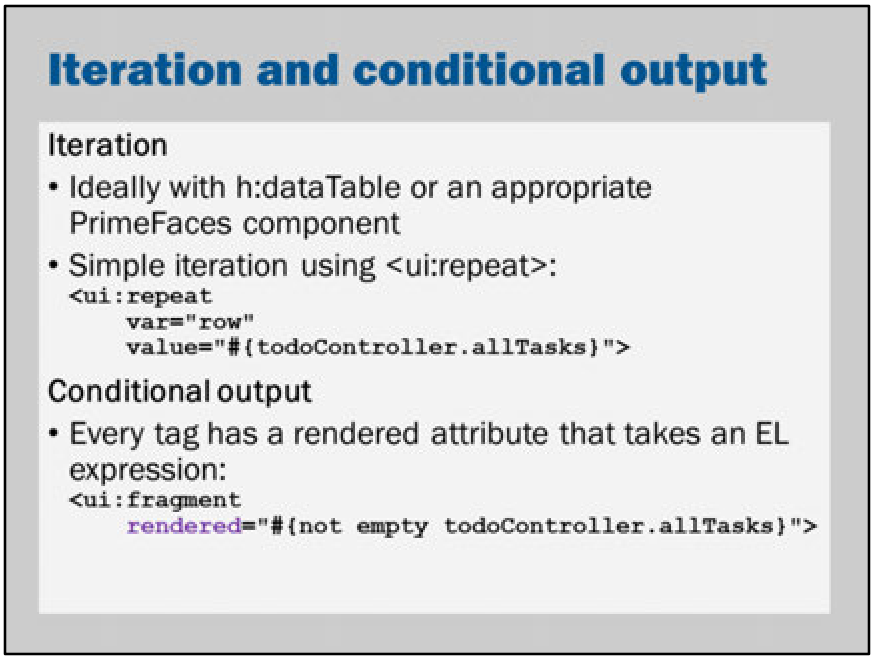
<h:commandButton>	<h:form>
<h:commandLink>	<h:inputFile>
<h:link>	<h:inputHidden>
<h:button>	<h:inputSecret>
<h:message>	<h:inputText>
<h:messages>	<h:inputTextArea>
<h:outputText>	<h:selectBooleanCheckbox>
<h:outputFormat>	<h:selectManyCheckbox>
<h:outputLabel>	<h:selectManyListbox>
<h:outputLink>	<h:selectManyMenu>
<h:dataTable>	<h:selectOneListbox>
<h:column>	<h:selectOneMenu>
<h:panelGrid>	<h:selectOneRadio>
<h:panelGroup>	<h:graphicImage>

These are the core JSF components for generating HTML:

Most of these are self-explanatory.

<h:message> and <h:messages> are used to show validation/error messages.

<h:dataTable> and <h:panelGrid> provide layout functionality.



Iteration and conditional output

Iteration

- Ideally with h:dataTable or an appropriate PrimeFaces component
- Simple iteration using <ui:repeat>:

```
<ui:repeat
     var="row"
     value="#{todoController.allTasks}">
```

Conditional output

- Every tag has a rendered attribute that takes an EL expression:

```
<ui:fragment
     rendered="#{not empty todoController.allTasks}">
```

In JSF, it is possible to iterate over data.
Generally speaking, it is better to use a specialized component that does the iteration for you.

For example, in JSP you would do something like this:

```
<table>
<% for (String row : todoController.getAllTasks()) {
%>
<tr>
   <td><%= row %></td>
</tr>
<%
}
%>
</table>
```

In contrast, in JSF it is better to use a component that matches the problem:

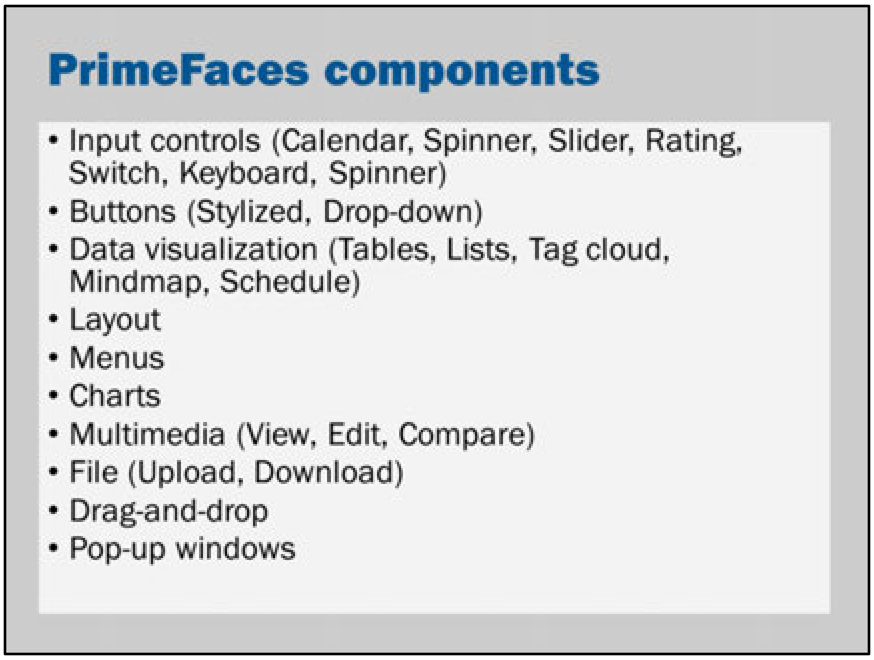
```
<h:dataTable var="row" value="#{todoController.allTasks}">
  <h:column>
```

```
    <h:outputText value=“#{row}”/>
  </h:column>
</h:dataTable>
```

However, if you do want to simply repeat something, then you can use the ui:repeat.

Here's how you might use iteration to show a bullet-list:

```
<ui:fragment rendered="#{not empty todoController.allTasks}">
  <ul>
    <ui:repeat var="row" value="#{todoController.allTasks}">
      <li>
        <h:outputText value="#{row}"/>
      </li>
    </ui:repeat>
  </ul>
</ui:fragment>
```



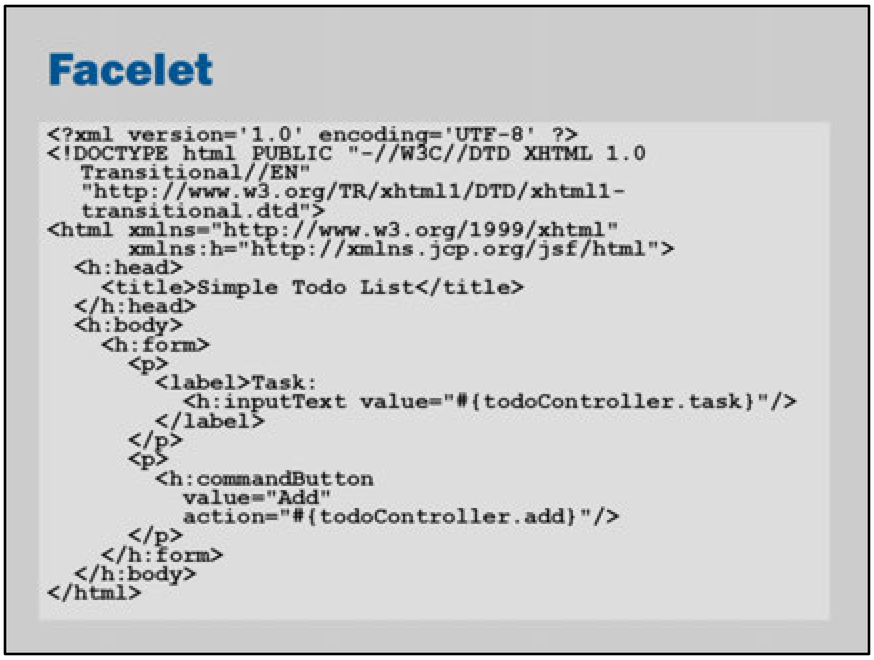
PrimeFaces components

- Input controls (Calendar, Spinner, Slider, Rating, Switch, Keyboard, Spinner)
- Buttons (Stylized, Drop-down)
- Data visualization (Tables, Lists, Tag cloud, Mindmap, Schedule)
- Layout
- Menus
- Charts
- Multimedia (View, Edit, Compare)
- File (Upload, Download)
- Drag-and-drop
- Pop-up windows

You aren't limited to the core JSF components.
There are several independent JSF component libraries.
One of the newer, more popular libraries is PrimeFaces.
It provides many many components for use in your application.

PrimeFaces can be very handy if you need a calendar/date input.
Rather than using a text field, PrimeFaces can create a JQuery-enabled date-picker.

You can see PrimeFaces information, including a "showcase" of components here:
http://primefaces.org/

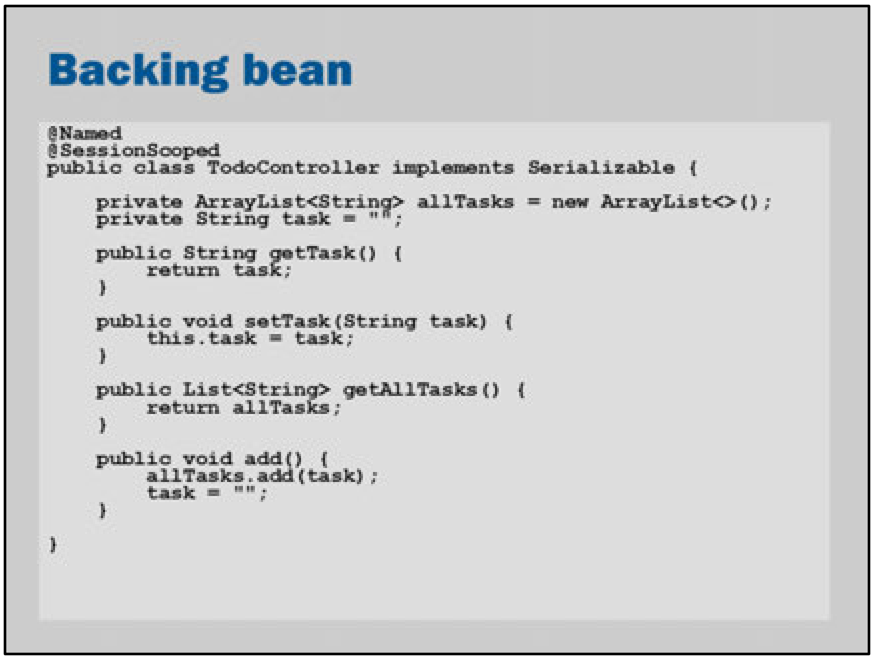


Facelet

```
<?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://xmlns.jcp.org/jsf/html">
  <h:head>
    <title>Simple Todo List</title>
  </h:head>
  <h:body>
    <h:form>
      <p>
        <label>Task:
          <h:inputText value="#{todoController.task}"/>
        </label>
      </p>
      <p>
        <h:commandButton
          value="Add"
          action="#{todoController.add}"/>
      </p>
    </h:form>
  </h:body>
</html>
```

This is the same slide as before…

Our Facelet uses Expression Language expressions.
These expressions refer to properties and methods on a Java Bean called a “backing bean”.



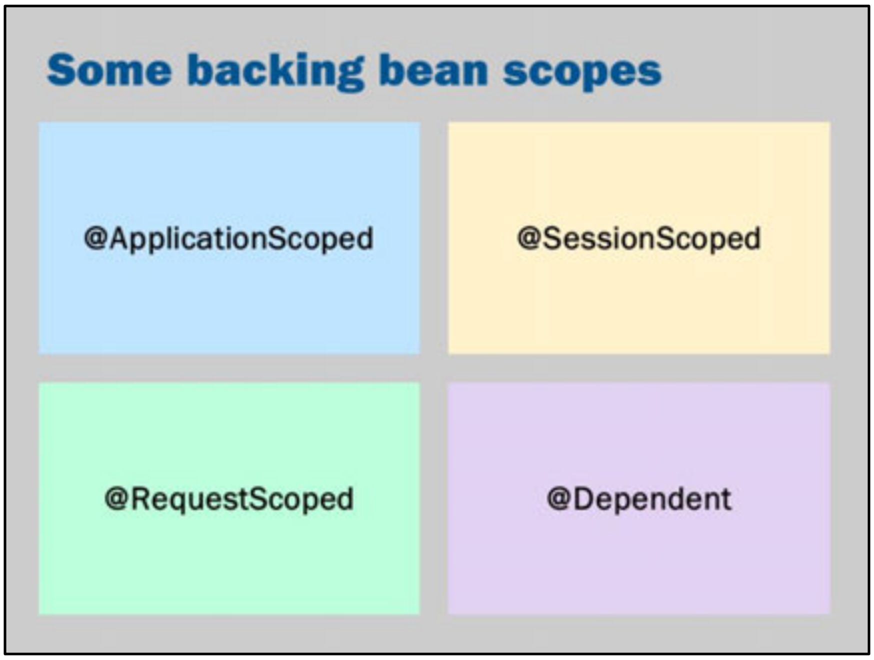
Backing bean

```
@Named
@SessionScoped
public class TodoController implements Serializable {

    private ArrayList<String> allTasks = new ArrayList<>();
    private String task = "";

    public String getTask() {
        return task;
    }

    public void setTask(String task) {
        this.task = task;
    }

    public List<String> getAllTasks() {
        return allTasks;
    }

    public void add() {
        allTasks.add(task);
        task = "";
    }

}
```

The backing bean is our Java code that implements the functionality of our JSF application.

The key thing that makes this a backing bean is the first line: @Named
@Named tells Java EE that this class can be referred to by name.
In this case, the TodoController class is referred to by the name “todoController”.
The naming rule, like many name convesion rules in Java, is to take the name and make the first letter lowercase.

The @SessionScoped tells JSF when to create the backing bean.
A @SessionScoped bean will be created once per user.
In the tutorial, we use @RequestScoped beans that are created with each request.



Some backing bean scopes

@ApplicationScoped	@SessionScoped
@RequestScoped	@Dependent

We have seen @RequestScoped many times in JavaServer Faces.
This creates a new instance every time a new request comes in.

@ApplicationScoped creates only one instance across the entire application.

@SessionScoped creates one instance per user.

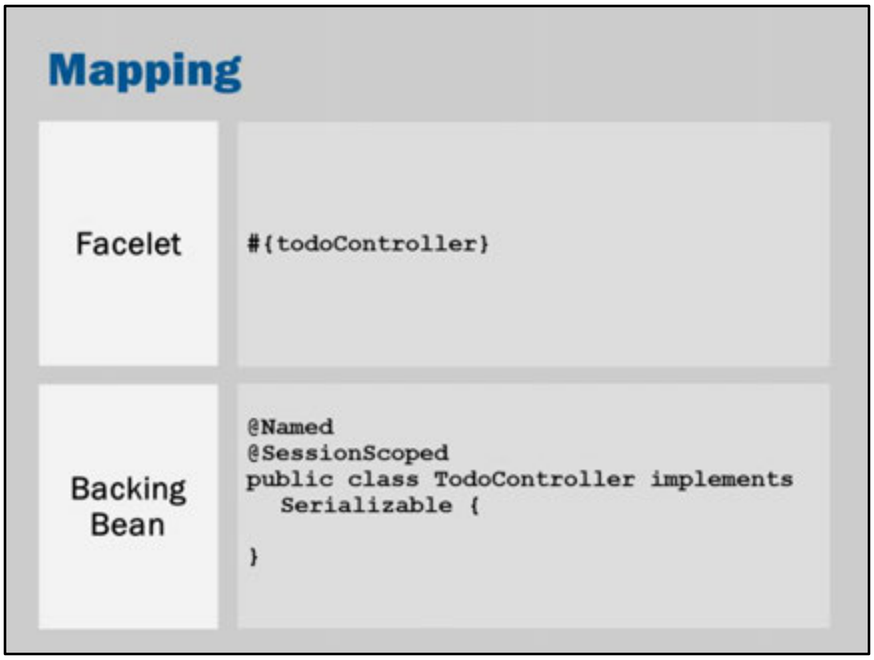
@Dependent creates a new instance every time it is used. The scope is "dependent" on the scope of the class that it was injected into.

Advanced note:
There are two other scopes in common use:

@ViewScoped creates one instance for a single page viewed by a user (so it is a bit like session scoped, but tied to the particular page that the user is looking at – and if they've got to separate browser windows open, then that's two separate views)

@ConversationScoped works like session scoped, but there is a separate instance for each browser tab and it can be started and finished by the application.

More information:
https://docs.jboss.org/weld/reference/1.0.0/en-US/html/scopescontexts.html

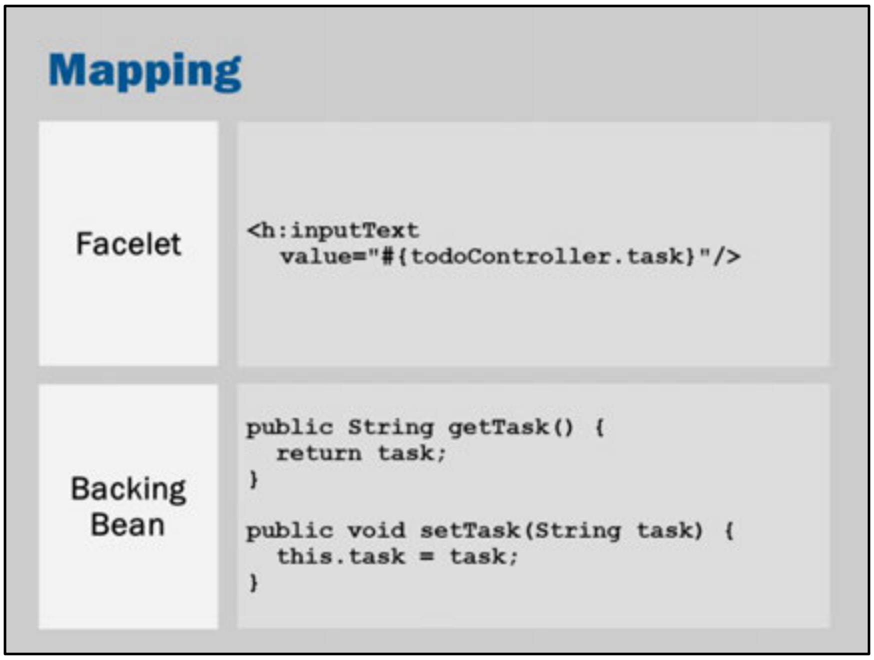


Mapping

Facelet	`#{todoController}`
Backing Bean	see code below

```
@Named
@SessionScoped
public class TodoController implements
   Serializable {

}
```

In our Facelet, we have an Expression Language (EL) expression “#{todoController}”. The name “todoController” is found by looking for a Java Bean annotated with @Named.



Mapping

Facelet	`<h:inputText value="#{todoController.task}"/>`
Backing Bean	`public String getTask() { return task; } public void setTask(String task) { this.task = task; }`

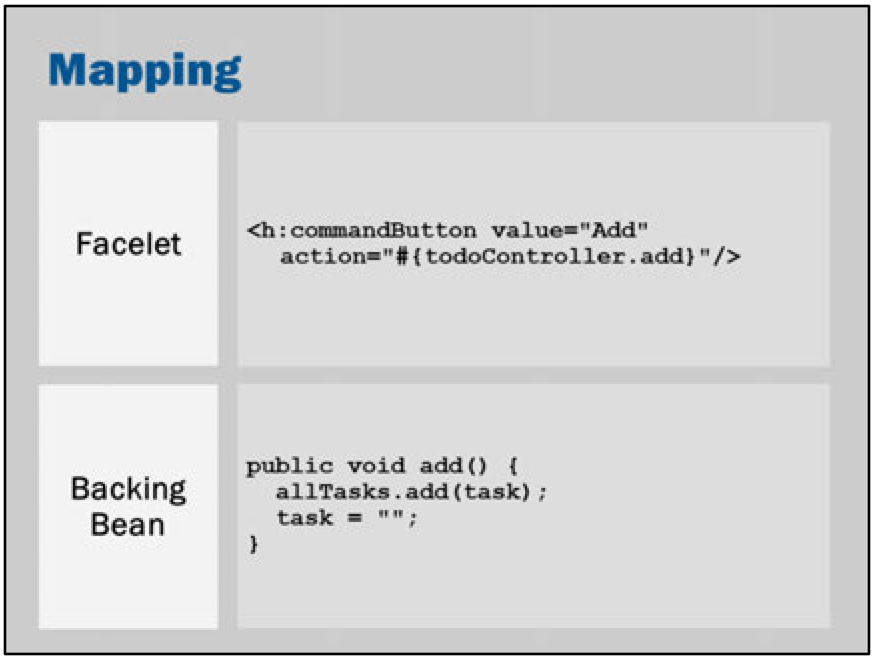
```
<h:inputText
   value="#{todoController.task}"/>
```

```
public String getTask() {
  return task;
}

public void setTask(String task) {
  this.task = task;
}
```

The EL expression #{todoController.task} is mapped to our Java in two ways:

When we're generating the page, JSF will call todoController.getTask() to retrieve the current value of the task.
This current value will be shown as the default value in the text input.

When the user clicks on a button to submit the page, JSF will call todoController.setTask() to save the user input into the backing bean.



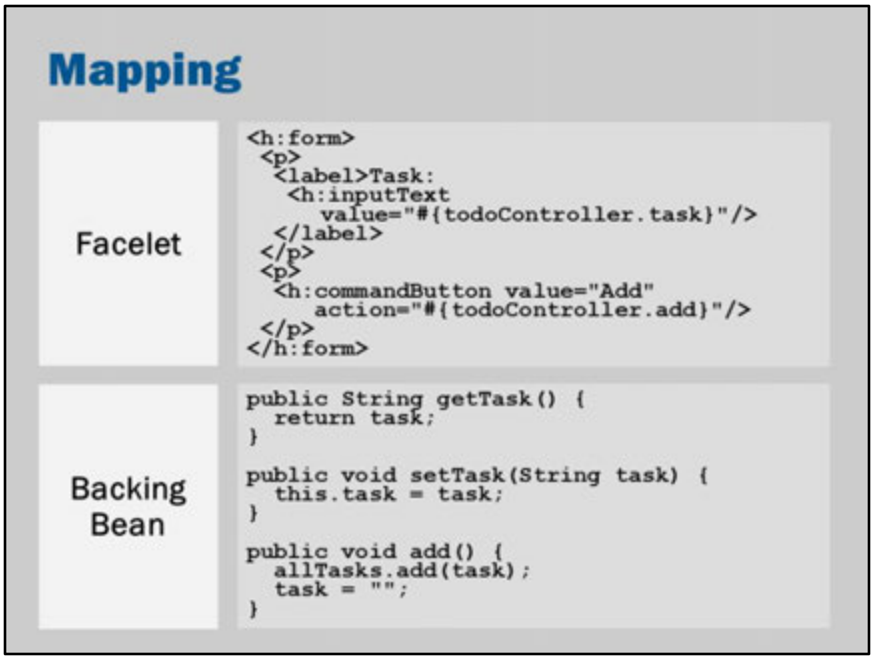
Mapping

Facelet	`<h:commandButton value="Add" action="#{todoController.add}"/>`
Backing Bean	`public void add() { allTasks.add(task); task = ""; }`

```
<h:commandButton value="Add"
   action="#{todoController.add}"/>
```

```
public void add() {
  allTasks.add(task);
  task = "";
}
```

On a command button, the action is an EL expression that refers to a method. The method is called when the user clicks on the button.



Mapping

Facelet

```
<h:form>
  <p>
   <label>Task:
    <h:inputText
       value="#{todoController.task}"/>
   </label>
  </p>
  <p>
   <h:commandButton value="Add"
      action="#{todoController.add}"/>
  </p>
</h:form>
```

Backing Bean

```
public String getTask() {
  return task;
}

public void setTask(String task) {
  this.task = task;
}

public void add() {
  allTasks.add(task);
  task = "";
}
```

This is the essential code in the view and the backing bean.

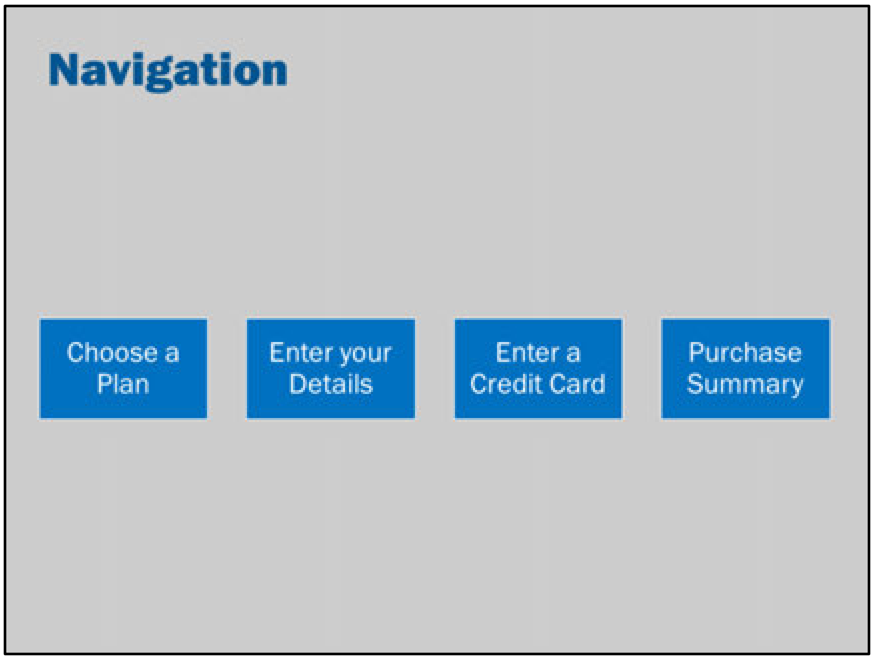
What would happen when we first request this page?

JSF would generate html for each of the tags.
When it gets to the <h:inputText> it would call getTask() to get the current value.
When it gets to the <h:commandButton> it wouldn't call any function… it would just 'remember' to call that function when the user click on it.

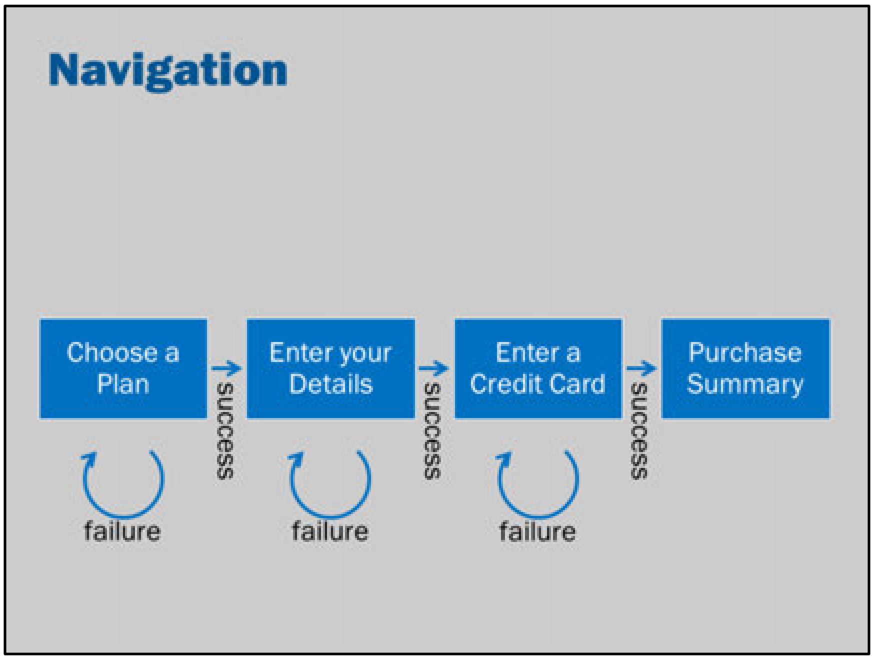
What would happen when the user clicks on the submit button?

The browser would submit the form.
JSF would go through the view and call setters and actions as appropriate.
When it gets to <h:inputText> it would call setTask(task) to set the value the user entered.
When it gets to the <h:commandButton> it would call the add() function on the backing bean.

Note: While I have suggested that JSF works from top-to-bottom, this isn’t technically correct. It works in phases. It will call all the setters before calling any of the actions. This “lifecycle” is covered in the next few slides.


Navigation

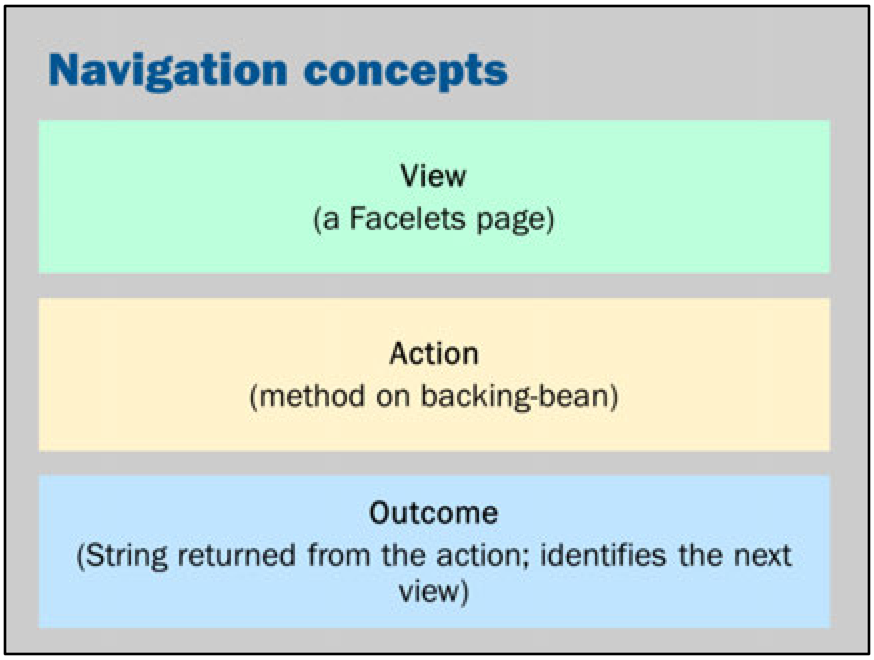
To understand navigation, consider an example of an application with multiple stages in signing up for an account.



The arrows express navigation between screens.

JSF allows us to directly represent those navigation rules.
Some benefits:

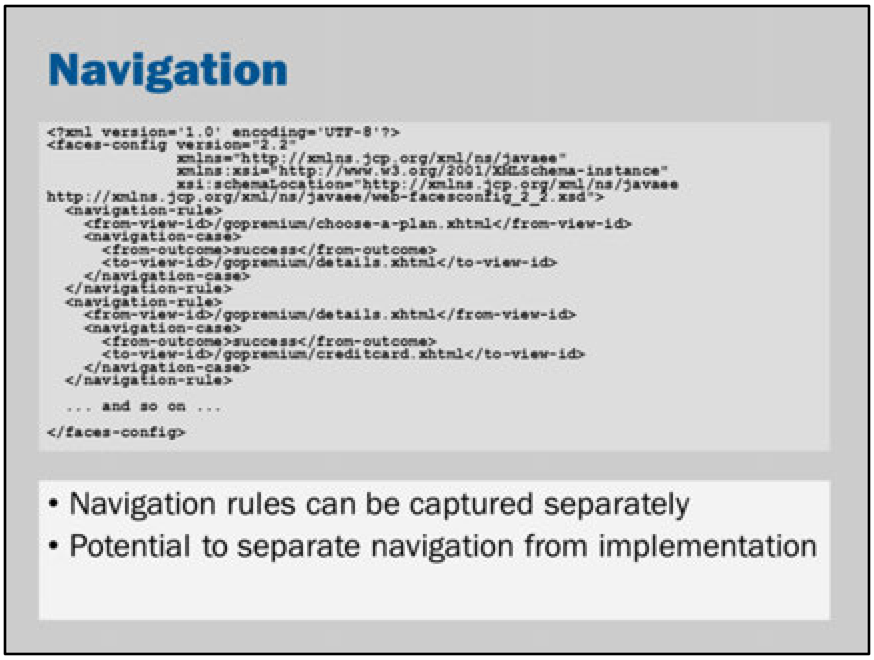
- Ability to easily change the navigation (e.g., In a payment workflow, should we get them to select a payment plan before or after collecting their billing address? By having navigation separate, we can change the order without changing the code.)
- Ability to change the names and/or paths of files. We can change the location of files and update the navigation rules rather than needing to change the code (or, perhaps, identify all places that we need to change the code).
- Ability to have actions return 'meaningful' responses (such as "success" or "failure"). JSF can then map these responses onto appropriate pages (i.e., on success, then go back to the main menu).



A view is a Facelets page.
It is what renders HTML for the user (and is also used to process the incoming request).

An action is called when the user clicks a button in the application.
The action is implemented as a method on the backing bean.

The action method needs to return a value.
This value is called an outcome and it is a String that identifies the next view to use.



Navigation

```
<?xml version='1.0' encoding='UTF-8'?>
<faces-config version="2.2"
              xmlns="http://xmlns.jcp.org/xml/ns/javaee"
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-facesconfig_2_2.xsd">
  <navigation-rule>
    <from-view-id>/gopremium/choose-a-plan.xhtml</from-view-id>
    <navigation-case>
      <from-outcome>success</from-outcome>
      <to-view-id>/gopremium/details.xhtml</to-view-id>
    </navigation-case>
  </navigation-rule>
  <navigation-rule>
    <from-view-id>/gopremium/details.xhtml</from-view-id>
    <navigation-case>
      <from-outcome>success</from-outcome>
      <to-view-id>/gopremium/creditcard.xhtml</to-view-id>
    </navigation-case>
  </navigation-rule>

  ... and so on ...

</faces-config>
```

- Navigation rules can be captured separately
- Potential to separate navigation from implementation

In JSF, navigation can be expressed using an XML file format.
Refer to the tutorial for a more detailed example.

Consider the navigation rule:

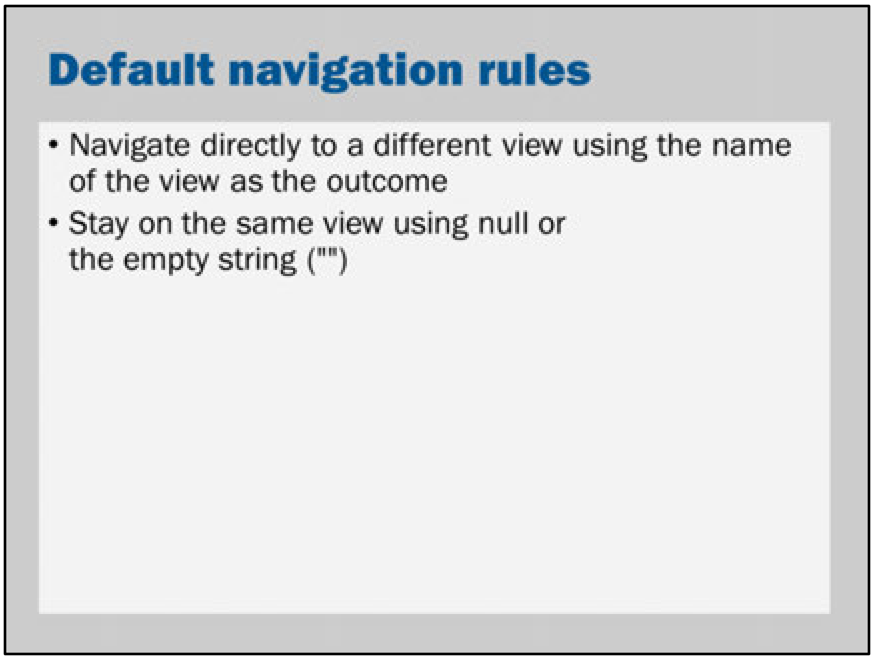
```
<navigation-rule>
   <from-view-id>/gopremium/choose-a-plan.xhtml</from-view-id>
   <navigation-case>
     <from-outcome>success</from-outcome>
     <to-view-id>/gopremium/details.xhtml</to-view-id>
   </navigation-case>
 </navigation-rule>
```

This says that when you're the choose-a-plan view…
<from-view-id>/gopremium/choose-a-plan.xhtml</from-view-id>

…then if the outcome "success" is encountered…
 <from-outcome>success</from-outcome>

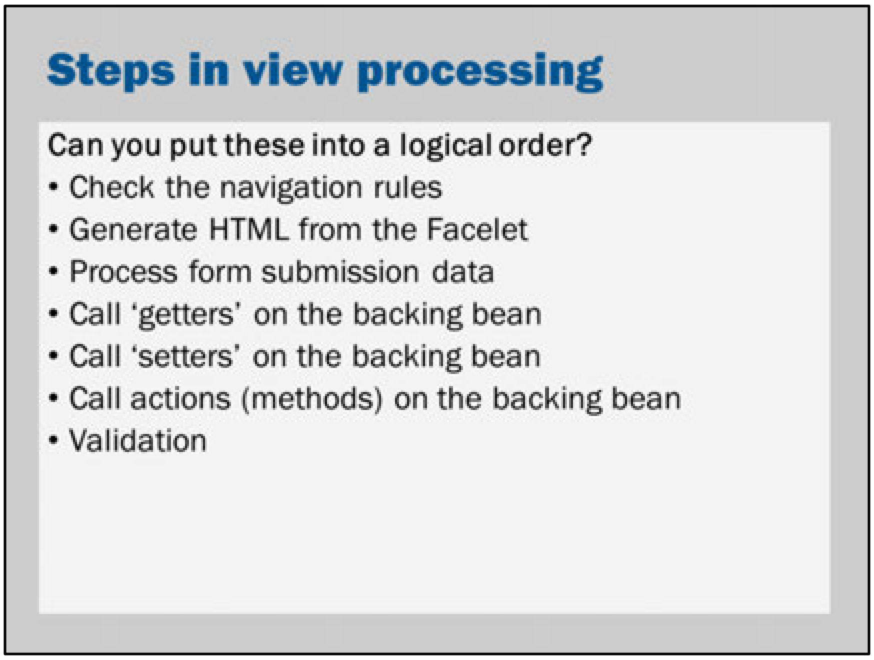
…then the application should show the details view…

```
<to-view-id>/gopremium/details.xhtml</to-view-id>
```



Default navigation rules

- Navigate directly to a different view using the name of the view as the outcome
- Stay on the same view using null or the empty string ("")

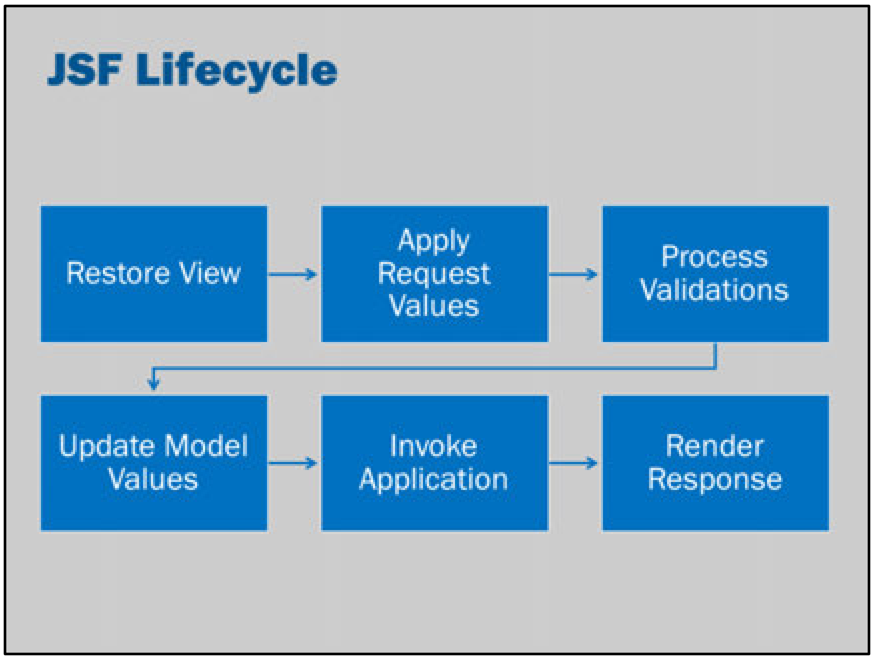


Steps in view processing

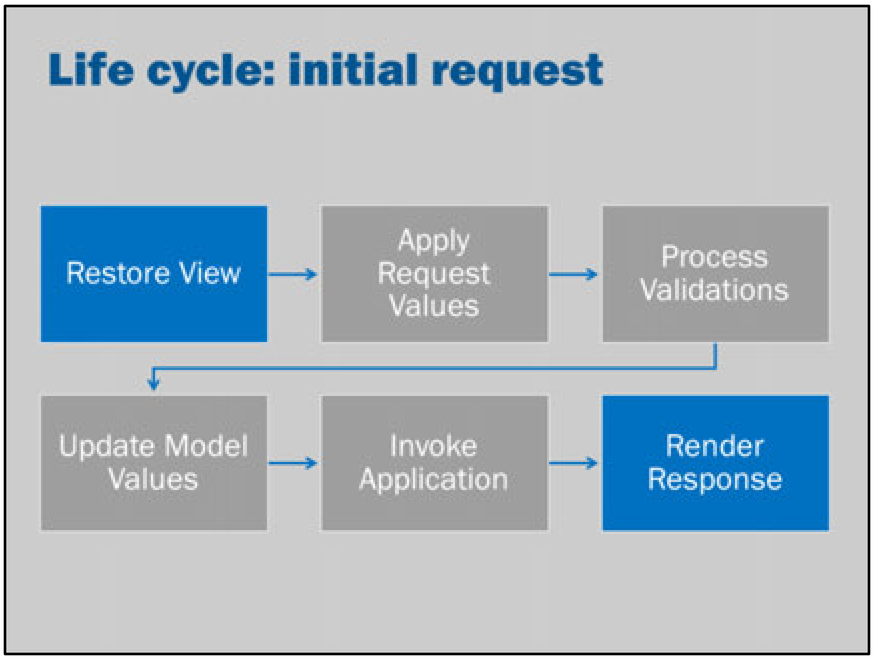
Can you put these into a logical order?

- Check the navigation rules
- Generate HTML from the Facelet
- Process form submission data
- Call ‘getters’ on the backing bean
- Call ‘setters’ on the backing bean
- Call actions (methods) on the backing bean
- Validation

JSF lifecycle



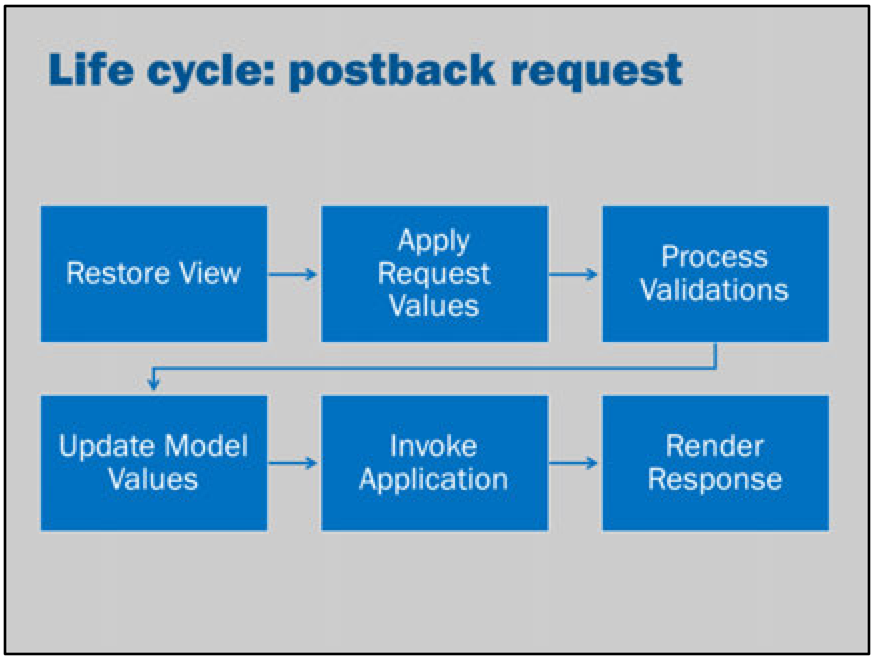
We have informally considered the process that JSF follows in handling a request.
This process is fully defined in the specifications.
This is referred to as a lifecycle.



When an initial request comes in, JSF has a simple two-stage process.

Restore View:
There is no view to restore, so JSF creates an empty view.

Render Response:
Then JSF fills the view with the components in the page: the component tree and event handlers.
The components then generate their HTML to respond to the user.


Life cycle: postback request

$$\begin{tikzcd}
\text{Restore View} \arrow[r] & \text{Apply Request Values} \arrow[r] & \text{Process Validations} \arrow[dll] \\
\text{Update Model Values} \arrow[r] & \text{Invoke Application} \arrow[r] & \text{Render Response}
\end{tikzcd}$$

In subsequent postbacks, JSF follows a more extensive lifecycle:

Restore View:
JSF stores the view of the page: all the component tree and event handlers.

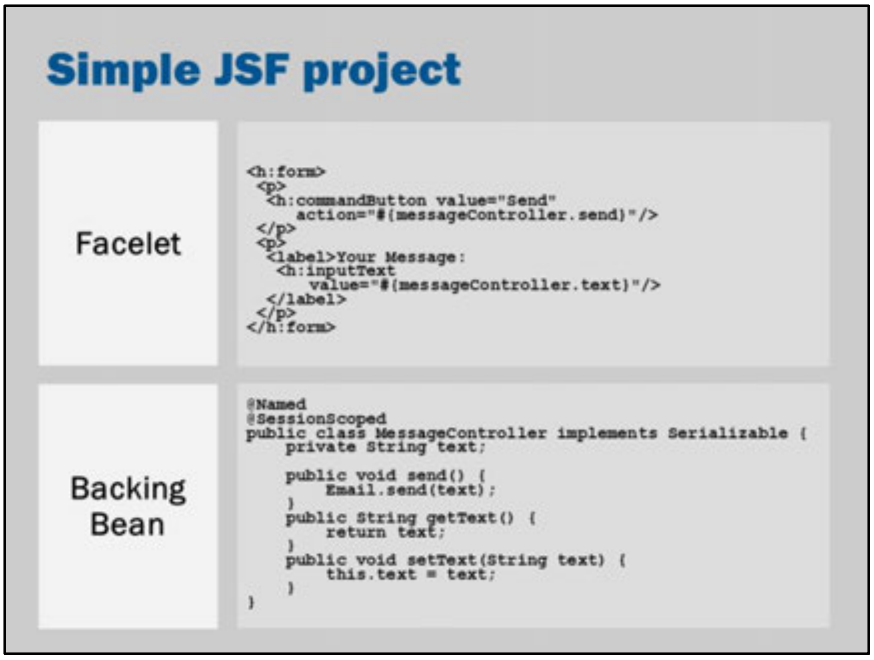
Apply Request Values:
The components on the page decode their request parameters – i.e., they retrieve parameters posted to the page.
The values are saved, temporarily, inside the component.

Process Validations:
All validators are called.
In the even of a validation error, the application skips ahead to render response.

Update Model Values:
The request values are then saved to the corresponding model.
That is, the values in the second phase are "written" to the backing bean and other model objects.

Invoke Application:
Application level-events are processed: i.e., the actions on the page are invoked.

Render Response:
The components generate their HTML to respond to the user.



Simple JSF project

Facelet

```
<h:form>
 <p>
  <h:commandButton value="Send"
     action="#{messageController.send}"/>
 </p>
 <p>
  <label>Your Message:
   <h:inputText
      value="#{messageController.text}"/>
  </label>
 </p>
</h:form>
```

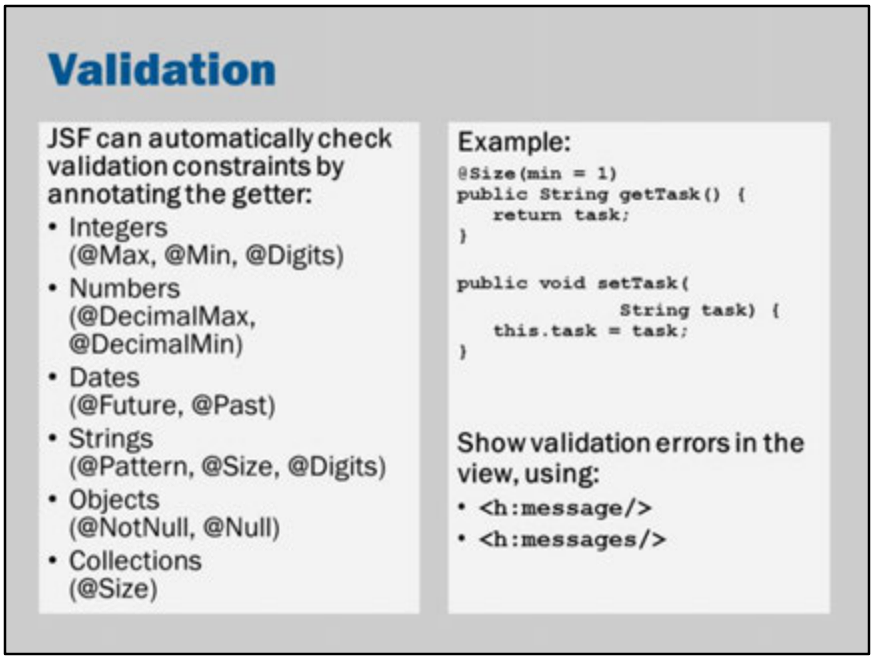
Backing Bean

```
@Named
@SessionScoped
public class MessageController implements Serializable {
    private String text;

    public void send() {
        Email.send(text);
    }
    public String getText() {
        return text;
    }
    public void setText(String text) {
        this.text = text;
    }
}
```

This is a simple JSF file that sends an email.

It has a text box (h:inputText) that is bound to the "text" property of the MessageController.
That is, it will call getText() to get the current value of the text to show the user and it will call setText(…) when the user posts back data when they click send.

The Send button is bound to the action method send() on the MessageController.
When the Send button is clicked, the form data will be posted back then (after calling setText to process the input text box) JavaServer Faces will call send().



Validation

JSF can automatically check validation constraints by annotating the getter:

- Integers (@Max, @Min, @Digits)
- Numbers (@DecimalMax, @DecimalMin)
- Dates (@Future, @Past)
- Strings (@Pattern, @Size, @Digits)
- Objects (@NotNull, @Null)
- Collections (@Size)

Example:

```
@Size(min = 1)
public String getTask() {
   return task;
}

public void setTask(
              String task) {
   this.task = task;
}
```

Show validation errors in the view, using:

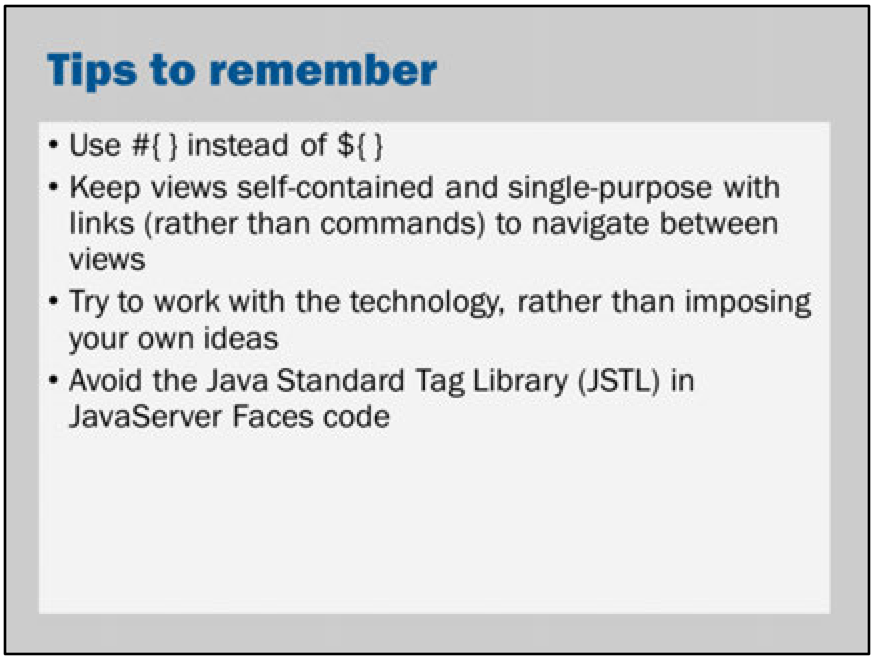
- `<h:message/>`
- `<h:messages/>`

One important step in this lifecycle is the “Process Validations”.
This is where JSF checks that the user input is valid.

```
@Size(min=1)
public String getTask() {
  return task;
}
```

Just by adding this single annotation enables JSF to automatically check that the input is correct and valid.
The annotation enforces that the user enters something into the task input text box.

JSF checks for validity by looking for these annotations on the properties of a class.
Simply adding the annotation to the getter is enough to get Facelets to perform these validation checks.



Tips to remember

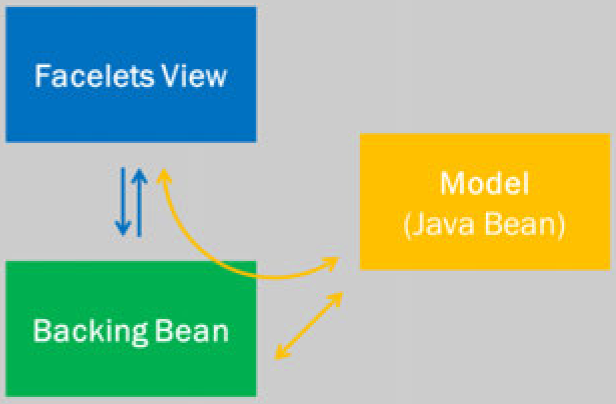
- Use #{ } instead of ${ }
- Keep views self-contained and single-purpose with links (rather than commands) to navigate between views
- Try to work with the technology, rather than imposing your own ideas
- Avoid the Java Standard Tag Library (JSTL) in JavaServer Faces code

These are a few more tips.

Warning! The JSTL will work in JSF. However, the JSTL is not designed with the JSF lifecycle in mind. There can be very complex interactions between the two that you may find very difficult to diagnose. Even though the JSTL can be used, I strongly recommend against it.

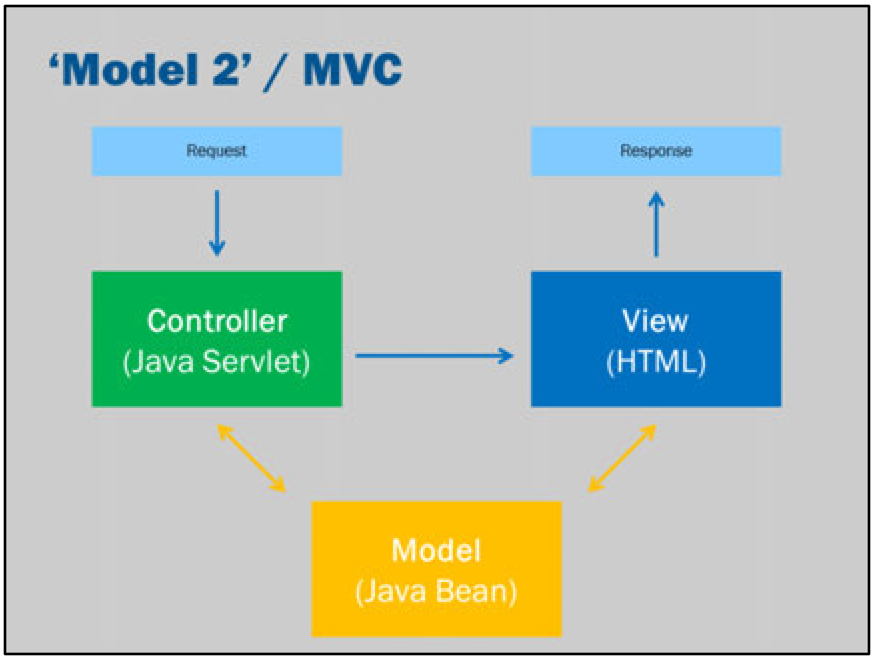
Bonus slides

Facelets applications



This image depicts how JavaServer Faces applies the MVC pattern.

We have views.
They are connected to backing beans.
The backing beans should make use of model classes.
The view accesses the model via the backing bean.



How does JSF map on to the MVC architecture?

MVC doesn't have a precise definition. It is a design pattern. It can be adapted to fit the situation.

There are at least two ways to view JSF as an MVC architecture:

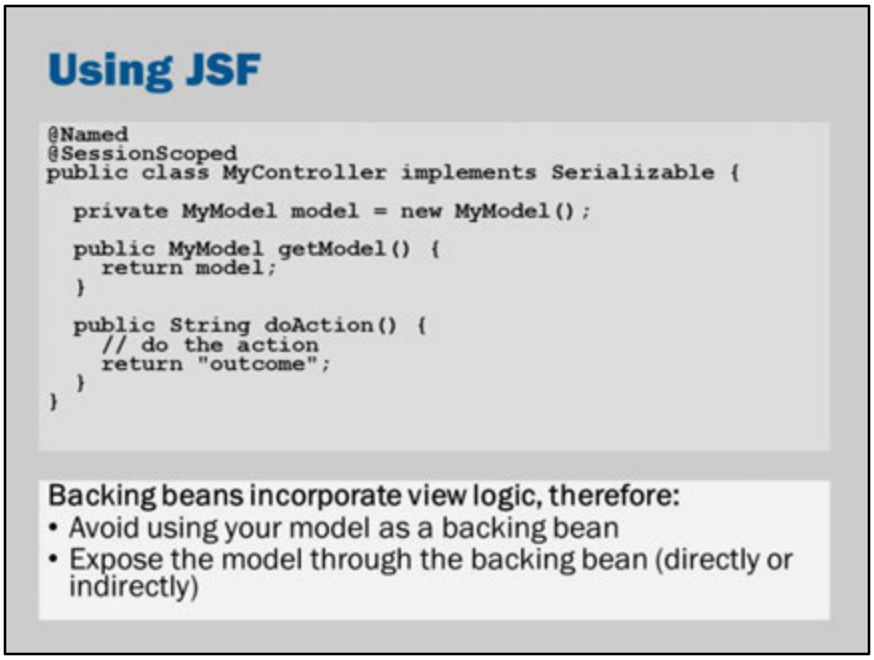
Controller = JSF Front Controller. i.e., JSF framework is the controller that handles the incoming requests.
Model = Backing Bean
View = Facelets page

or

Controller = Backing Bean
Model = Domain model classes that the backing bean uses
View = Facelets page

In the latter perspective, JSF is in the background. JSF is regarded as just part of the web server.

Personally, I prefer the second way of looking at how JSF relates to MVC.
However, there is no “right” answer.
Both are reasonable.



Using JSF

```
@Named
@SessionScoped
public class MyController implements Serializable {

  private MyModel model = new MyModel();

  public MyModel getModel() {
    return model;
  }

  public String doAction() {
    // do the action
    return "outcome";
  }
}
```

Backing beans incorporate view logic, therefore:
- Avoid using your model as a backing bean
- Expose the model through the backing bean (directly or indirectly)

JSF is a powerful and sophisticated framework.
However, it expects that you use it in certain ways.
If you don't use it in the "JSF way", its features will get in the way.
If you use it in an unusual way, you will find yourself annoyed and frustrated.

This slide shows a general pattern for your backing bean.
This pattern works well with JSF.
The idea is keep your model separate from your backing beans.
Use the backing bean as a simple front-end to JSF.
Don't use backing beans as your model and don't use backing beans to do something complex.
Your backing bean should be just getters, setters (that correspond to EL expressions in the view) and methods (that correspond to actions expressed using EL).