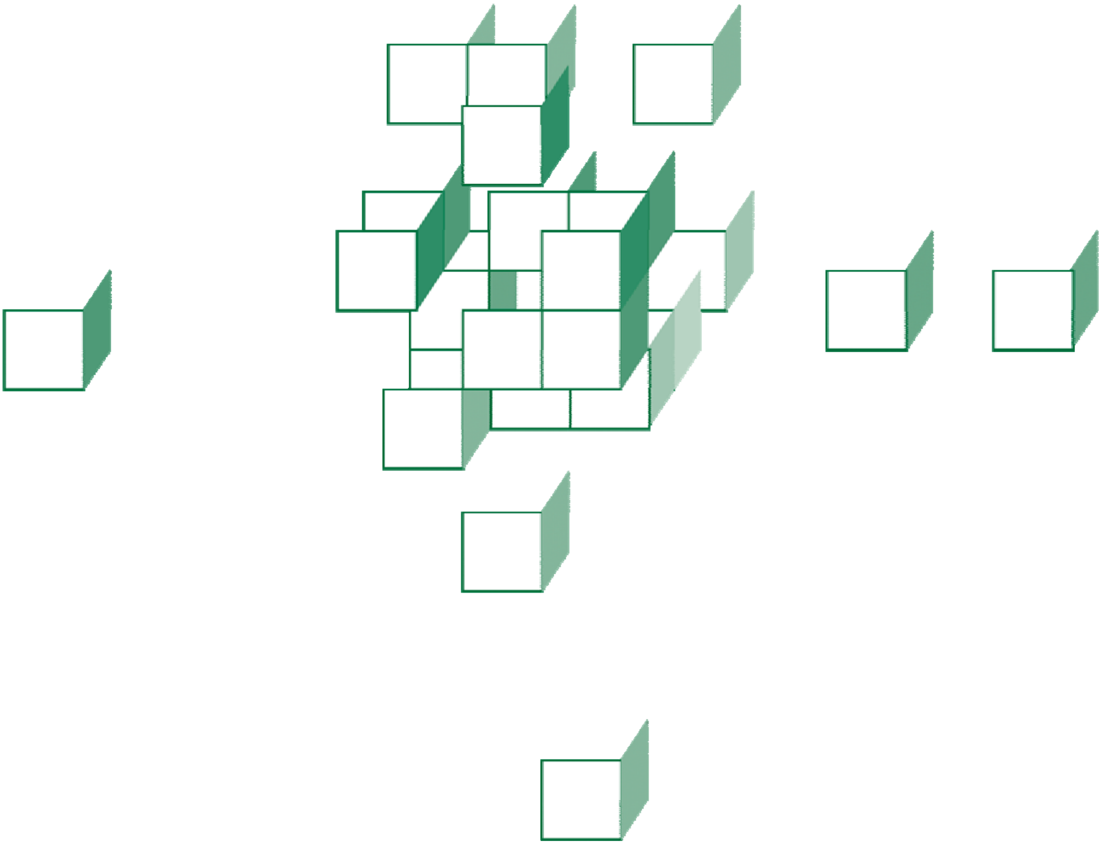
CHAPTER 12

VALIDATORS



OBJECTIVES

After completing "Validators," you will be able to:

- Describe the JSF validator architecture.
- Use standard validators and the **required** attribute to enforce basic input-validation rules.
- Define custom error messages, support error-message localization, and apply CSS styling specifically to error output.
- Implement custom validator classes and validation methods.
- Use UI tree navigation to enforce constraints involving multiple inputs.
- Take advantage of JSR-303 Bean Validation constraints when encountered on backing beans.

Validating Input

- **Well here we are, most of the way through our JSF course, and we've yet to acknowledge the sad fact that (shh):**

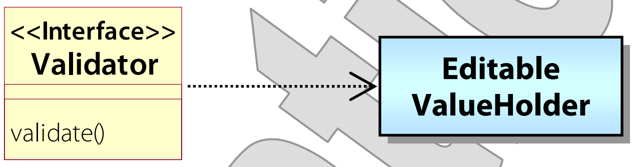
  > *Users don't always do what they're supposed to do.*

- **Most web applications devote a significant portion of their logic to validating user input:**
  - Did the user provide all the **required** information?
  - Are data of the correct **type** (number, boolean, date, etc.)?
  - Are given values in legal **ranges**?
  - Are values provided in correct numerical, alphabetical or chronological **order**?
  - Do values **match** where they're expected to match?
- **Validation is a form of error handling, and it is proactive, or eager: we seek out possible problems and report them immediately.**
  - This saves processing time and bandwidth.
  - It also tends to result in clearer reporting: would you rather get a message that one of the values you typed isn't legal, for a specific reason – or a **NullPointerException** from some faraway province of Java code that you probably didn't write?
- **Validation is also an important application-security tool, because here's another sad truth:**

  > *Users aren't always trying do what they're supposed to do; sometimes, they're trying to break into your system.*

The Validator Interface

- JSF defines the role of a **validator**: a component that is responsible for testing the validity of a component value.
- It encapsulates this basic responsibility in the **Validator** interface:

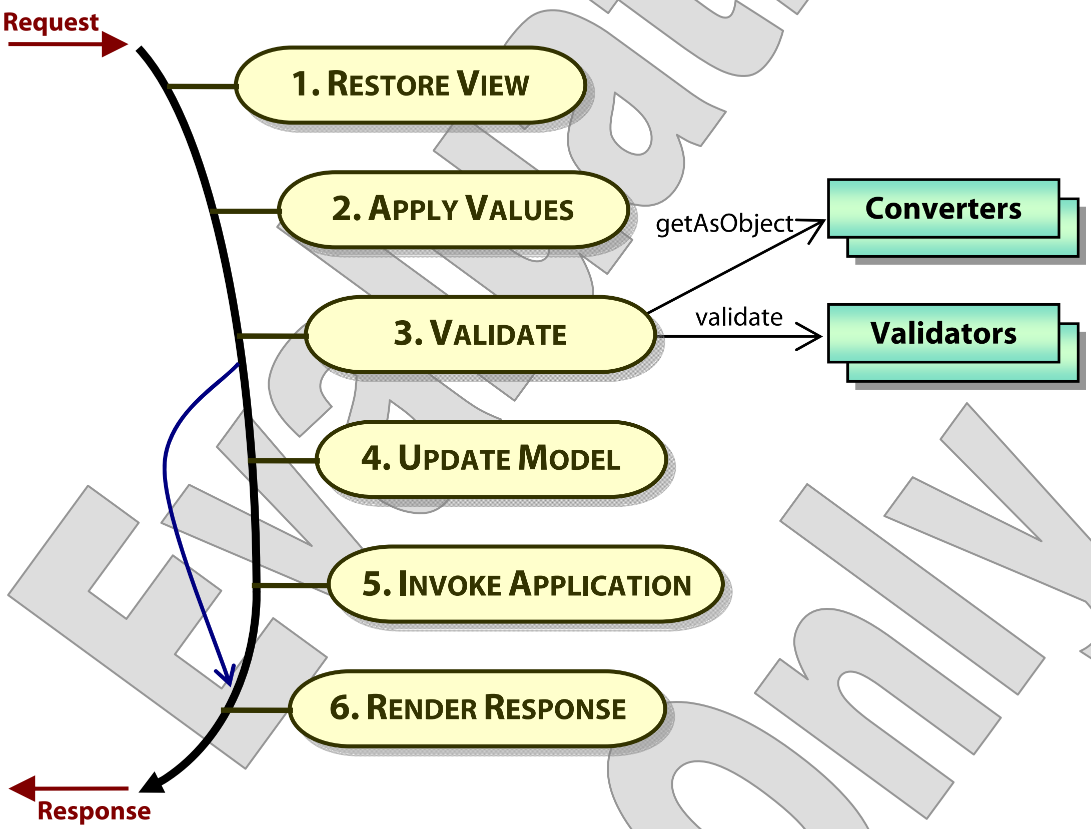


```
public interface Validator
{
  public void validate
      (FacesContext, UIComponent, Object value)
    throws ValidatorException;
}
```

- As with **Converter**, the context and component parameters often go unused, but they can come in handy for some purposes.
- Most validators just perform tests on the given **value**.
- If the value is not valid, the validator must throw a **ValidatorException**, which wraps an error message.

Handling Validity Errors

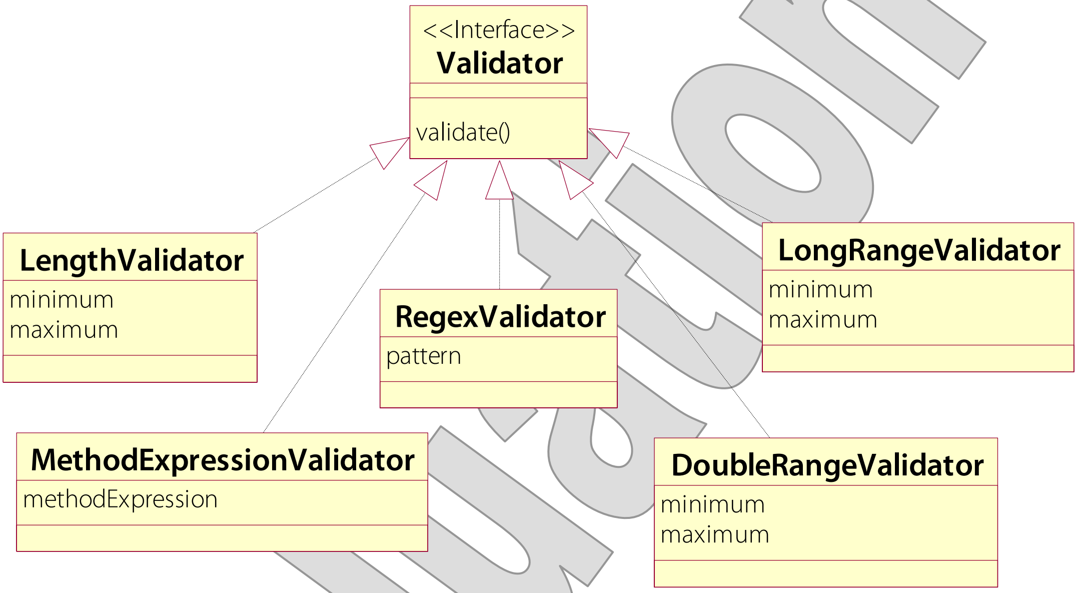
- Validators are registered on components and invoked during – you guessed it! – the Process Validations phase.
- If any **ValidatorExceptions** are caught during this phase, the associated component's **valid** property is set to false.
- At the end of the phase, if there are any invalid components, JSF bails out of the cycle, moving directly to Render Response.



- In fact the same sort of handling occurs on any conversion failures – different exceptions, different messages, same flow.

Standard Validators and Required Values

- JSF provides a handful of validators for the most common logic:



<<Interface>>
Validator
validate()

LengthValidator
minimum
maximum

RegexValidator
pattern

LongRangeValidator
minimum
maximum

MethodExpressionValidator
methodExpression

DoubleRangeValidator
minimum
maximum

  - **LengthValidator** constrains the length of the lexical representation – regardless of the data type.
  - The **range validators** allow you to set a minimum value, maximum value, or both, for integral or floating-point numbers.
  - **MethodExpressionValidator** is a way to trigger a method on a JavaBean and let it perform validation – this is one of three main ways in JSF to plug in custom validation logic.
  - The **RegexValidator** applies a given **pattern** as a regular expression; non-matching values are considered invalid.

JSF 2.0

- You may wonder at the absence of a **RequiredValidator**.
- This is such a common need that JSF makes it even simpler:
  - Setting **required** to **true** on any **EditableValueHolder** causes JSF to treat a null or blank value as invalid for the component.

Declaring Validation Rules in the View

- Set **required** to **true** to force a check for a non-empty value:

```
<h:inputText value="#{bean.prop}"
  required="true"
/>
```

- Include the appropriate core tag as a child of the component to assign a standard validator – you can assign multiple validators:

```
<h:inputText value="#{bean.fullName}" >
  <f:validateLength maximum="32" />
</h:inputText>

<h:inputText value="#{bean.age}" >
  <f:validateLongRange minimum="18" />
</h:inputText>

<h:inputText value="#{bean.probability}" >
  <f:validateDoubleRange minimum="0" maximum="1" />
  <f:validateLength maximum="6" />
</h:inputText>
```

- Note one frustrating limitation of the **DoubleRangeValidator**: it cannot be told to exclude a boundary value.
  - It will always work **inclusively** – that is, they will always allow your stated minimum or maximum value as a valid value.
  - This is fine for integer ranges, but we often find reason to find a floating-point number to be valid if it is, say, greater than a minimum (not greater than or equal to).
  - What if we want a positive number? Zero is the minimum but should be excluded. **DoubleRangeValidator** can't do that.

Required Fields

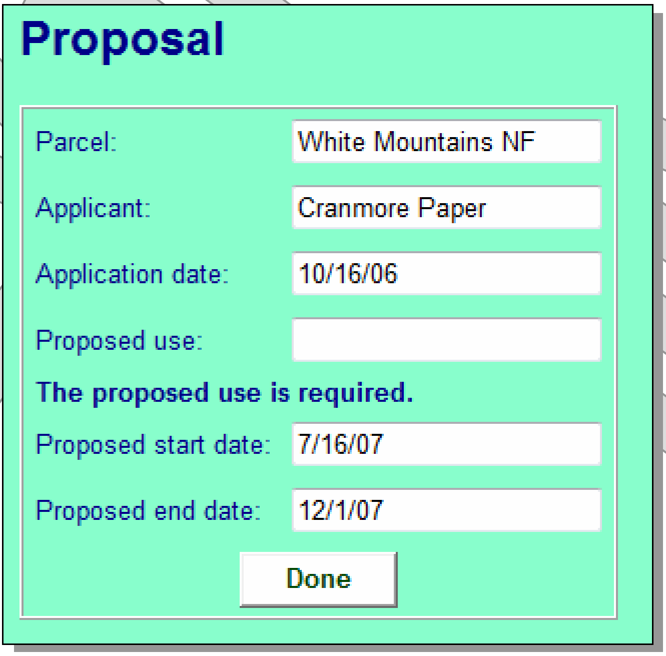
EXAMPLE

- **Examples/LandUse** provides a simple example of the use of the **required** attribute – see **docroot/detail.xhtml**:

```
<td>Parcel:</td>
<td>
  <h:inputText
    id="affectedParcel"
    label="affected parcel"
    value="#{DB.selectedProposal.affectedParcel}"
    required="true"
  />
</td>
```

- Any failure to provide required values results in an error message:

`http://localhost:8080/LandUse`



Proposal

Parcel:	White Mountains NF
Applicant:	Cranmore Paper
Application date:	10/16/06
Proposed use:	
The proposed use is required.	
Proposed start date:	7/16/07
Proposed end date:	12/1/07

Strong Passwords

DEMO

- **We'll begin a small case study for this chapter with a demonstration of constraining password length and format.**
  - Do your work in **Demos/Validation**.
  - The completed demo is in **Examples/Passwords/Step2**.
- **The starter code lays out a three-field form by which the user can register as a member of a website.**
- **It sets the required flag on each component, and uses `<h:messages>` as the simplest means of feeding error messages back to the user.**
  - See **docroot/register.xhtml**:

```
<f:view>
  <h:form>
    <h:messages />
    <table>
      <tr>
        <td>User name:</td>
        <td><h:inputText
              id="name"
              value="#{member.name}"
              required="true"
            /></td>
```

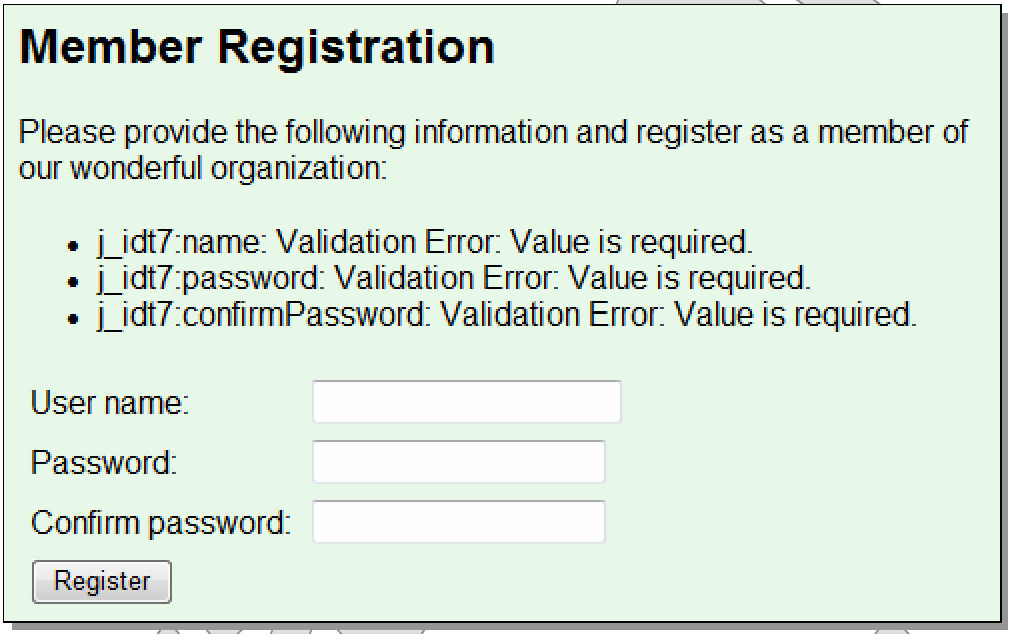
- **The command button will trigger navigation to a success.jsp, but whenever validation errors occur, JSF skips to rendering the response, and always with a null outcome, so the originating page is served again.**

```
<h:commandButton value="Register"
                 action="success" />
```

Strong Passwords

1. Build and test the starter application. If you fail to enter values, you see three not-very-friendly error messages:

```
http://localhost:8080/Passwords
```



Member Registration

Please provide the following information and register as a member of our wonderful organization:

- j_idt7:name: Validation Error: Value is required.
- j_idt7:password: Validation Error: Value is required.
- j_idt7:confirmPassword: Validation Error: Value is required.

2. Before we even get into validation, let's make those messages at least a little nicer. We already have **id** attributes on the individual text fields. But since we don't define an **id** for the **`<h:form>`**, JSF generates one for us. Define one now:

```
<h:form id="form" >
```

3. If you test again, with just that one change, you'll see the error messages all start with "form:" instead of "j_idt7".

Strong Passwords

DEMO

4. We can do even better by setting **label** attributes into the text fields themselves – as in:

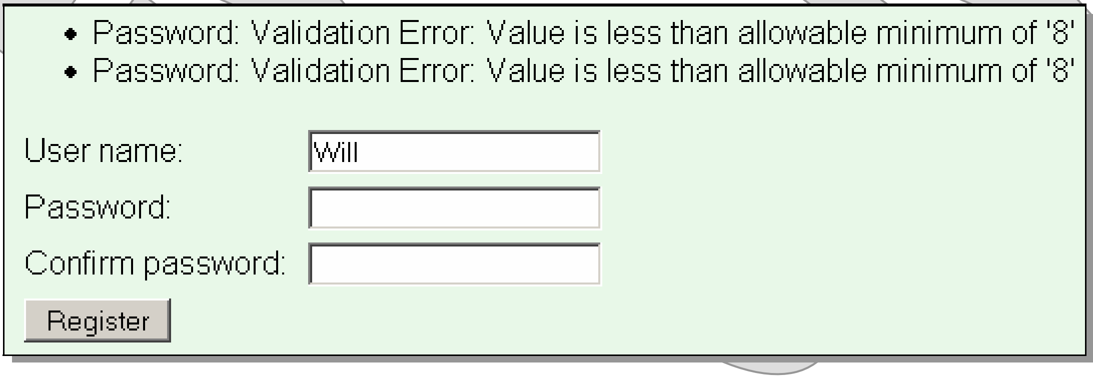
```
<h:inputSecret
  id="password"
  label="Password"
  value="#{member.confirmPassword}"
  required="true"
/>
```

5. Now the whole ID, for example “form:password”, becomes simply “Password” – test this now if you like, or see it in later steps.

6. Now set length constraints on each of the two password fields:

```
<h:inputSecret id="password" label="Password"
  value="#{member.confirmPassword}"
  required="true"
>
  <f:validateLength minimum="8" />
</h:inputSecret>
```

7. Build and test, and try providing shorter passwords:



- Password: Validation Error: Value is less than allowable minimum of '8'
- Password: Validation Error: Value is less than allowable minimum of '8'

User name: Will

Password:

Confirm password:

   – The previously-entered password value is cleared when the page is re-served, because this is an **`<h:inputSecret>`** component.

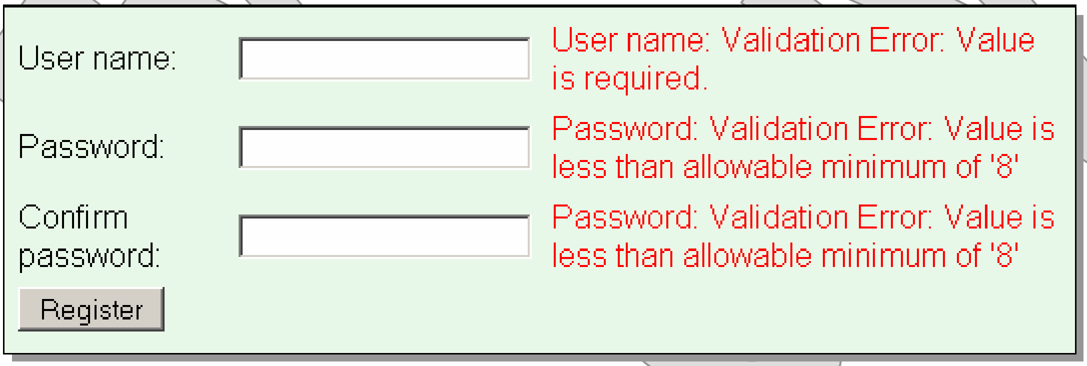
8. So we have our validation logic in place; let's clean up the presentation of error messages a bit. Get rid of the **`<h:messages>`** tag at the top of the form.

9. To each of the three rows of the table with input components, add a third cell that holds an **`<h:message>`** component. Call out the client ID for which you want to show messages, and set an **errorStyle**:

```
  <td>User name:</td>
  <td><h:inputText id="name" label="User name"
        value="#{member.name}"
        required="true"
      /></td>
  <td><h:message for="name"
                 errorClass="errorMessage" /></td>
</tr>
```

   – This style is already defined in **docroot/register.css**.

10. Build and test one last time, and see that messages are now visually connected to their subjects:



User name: | User name: Validation Error: Value is required.
Password: | Password: Validation Error: Value is less than allowable minimum of '8'
Confirm password: | Password: Validation Error: Value is less than allowable minimum of '8'

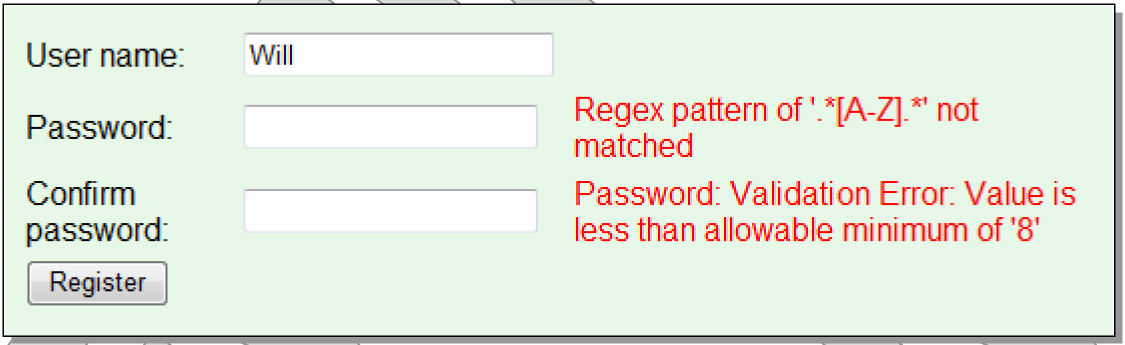
Strong Passwords

11. Finally, let's insist on stronger passwords by requiring at least one of each of a few character classes. Apply the following validators to each of the two password fields:

```
<f:validateRegex pattern=".*[A-Z].*" />
<f:validateRegex pattern=".*[a-z].*" />
<f:validateRegex pattern=".*[0-9].*" />
```

12. Build and test, and see that none of the following passwords will be found valid ...

`Provost`	`(no digit)`
`mypassword3`	`(no capital letter)`
`Aa1`	`(too short!)`



User name: Will

Password:

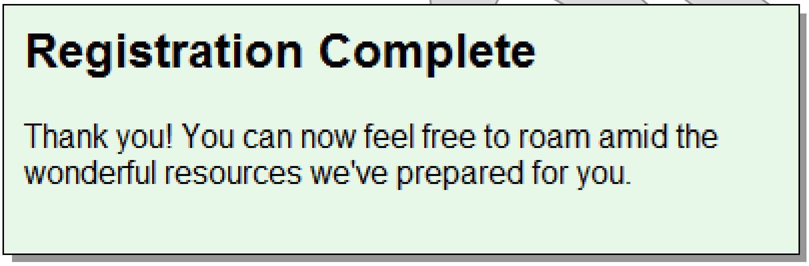
Confirm password:

Regex pattern of '.*[A-Z].*' not matched

Password: Validation Error: Value is less than allowable minimum of '8'

– ... while these will:

`Provost9`
`BIG000deal`

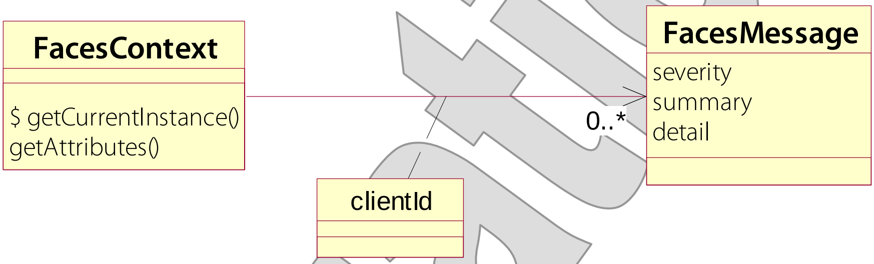


Registration Complete

Thank you! You can now feel free to roam amid the wonderful resources we've prepared for you.

Producing Error Messages

- Each **ValidatorException** wraps an instance of **FacesMessage**.
- These are collected during the Apply Request Values and Process Validations phases by the **FacesContext** object:



  - It is not a simple list or bag of messages, but a map.
  - The key is the **client ID** of the component with the problem.
  - And, it's not a simple map but a **map of lists of messages**.
  - This makes sense for what we need to do with error messages, because each field can encounter multiple problems:

firstName	**REQUIRED**
lastName	(no errors)
bankBalance	**BELOW MINIMUM**
	INVALID FRACTION
	INVALID CURRENCY

The FacesMessage Class

- **FacesMessage**, in turn, encapsulates three values: **severity**, **summary** message, and **detail** message.
- Message severity is a simple enumeration of possible levels – these are defined as instances of the inner class **FacesMessage.Severity**:

```
FacesMessage.SEVERITY_FATAL
FacesMessage.SEVERITY_ERROR
FacesMessage.SEVERITY_WARN
FacesMessage.SEVERITY_INFO
```

- Each message includes a pair of strings for summary and detail.
  - There is no formal distinction between these, and often they're identical.
  - The framework is just giving you some room to define shorter and longer versions of messages if you choose to do so.
  - The UI components that present messages can be tweaked to show one or the other: by default `<h:messages>` shows summaries and `<h:message>` shows details.

Message Keys

- Server-side code must produce error messages – whether that code is in the JSF framework, your application, or both.
- But error messages become visible to the user on the client side.
- This means that they must be localized – or at least that JSF must support localization of your messages.
- By default, the standard validators will use pre-defined message keys to generate their **FacesMessage** instances.
- Here are some of the most common validator keys:

```
javax.faces.component.UIInput.REQUIRED
javax.faces.validator.LengthValidator.MINIMUM
javax.faces.validator.LengthValidator.MAXIMUM
javax.faces.validator.LongRangeValidator.MINIMUM
javax.faces.validator.LongRangeValidator.MAXIMUM
javax.faces.validator.DoubleRangeValidator.MINIMUM
javax.faces.validator.DoubleRangeValidator.MAXIMUM
```

- See the JSF specification (section 2.5.2.4) for a complete list.
  - The great majority of standard keys are actually for type-conversion errors, and not specifically for validators.

Message Parameters

- **The message values all carry replaceable parameters – here's the message for required fields:**

```
{0}: Validation Error: Value is required
```

- **The rules for replacement are a little vague in the specification, but one thing is stated clearly:**
  - The **last** of the numbered parameters will be replaced with the component's **label**.
  - Other parameters, if present, will mean different things for different messages and validators.
  - It's usually obvious from the message what each parameter should mean; here's the message for minimum integer value:

```
javax.faces.validator.LongRangeValidator.MINIMUM =
  {1}: Validation Error: Value is greater than
  allowable maximum of "{0}"
```

Presenting Error Messages

- **The custom tags `<h:messages>` and `<h:message>` each take different approaches to rendering error messages onto a page.**
  - `<h:messages>` will render the summary value of every message in the context, in a simple bullet-list style.
  - `<h:message>` will render the detail value for the first message for a specific component – keyed by a client ID supplied by the **for** attribute – as raw text or as an HTML `<span>` to implement specific formatting.
- **It's possible to use both:**
  - `<h:messages>` at the top of the page
  - An `<h:message>` for each component, as a third column on the table

```
<f:view>
 <h:messages />
 <h:form>
   ...
     <h:inputText id="a" label="Semi-axis A"
       value="#{ellipsoid.a}"
       required="true" >
      <f:validateDoubleRange minimum="0.0000001"/>
     </h:inputText>
    </td>
    <td><h:message for="a" /></td>
   ...
```

  - It's also possible to use either **absolute or relative client IDs**; if relative they will be based on the nearest **NamingContainer** enclosing the `<h:message>` tag.

The errorStyle and errorClass Attributes

- Both of these tags support the **errorStyle** attribute – along with **fatalStyle**, **warnStyle**, and **infoStyle**.
  - Each attribute defines styling for a specific **message severity**.
- A second set of attributes are **fatalClass**, **errorClass**, **warnClass**, and **infoClass**.
  - Here the value is a CSS class, which will be defined on a (usually) separate stylesheet.
- Conversion and validation errors will exhibit error-level severity – this is **FacesMessage.SEVERITY_ERROR**.
- So, use **errorClass** to indicate presentation styling for your validation and conversion error messages:

```
<h:message
  for=":form:firstName"
  errorClass="errorMessage"
/>
```

  - The error-message class in the associated stylesheet might be:

```
.errorMessage
{
  color: red;
  font-weight: bold;
}
```

Custom Messages

DEMO

- **Picking up where we left off with our user-registration page, let's further fine-tune the message output with custom messages for required fields and string length.**
  - Do your work in **Demos/Messages**, or continue your work in **Demos/Validation**.
  - The completed demo is in **Examples/Passwords/Step3**.

1. Open the configuration file and declare an `<application>` config with one resource bundle:

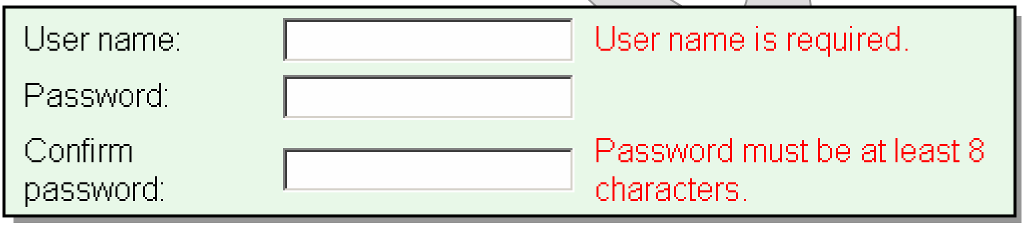
```
<application>
  <message-bundle>Resources</message-bundle>
</application>
```

2. Create a file **docroot/WEB-INF/classes/Resources.properties**, with the following message keys and values:

```
javax.faces.component.UIInput.REQUIRED=
  {0} is required.
javax.faces.validator.LengthValidator.MINIMUM=
  {1} must be at least {0} characters.
```

   - Note that there should be no line breaks between the key, the equals sign, and the value; the line breaks above are used to format the information for the coursebook.

3. Build and test again, and see your messages in play:



User name:		User name is required.
Password:		
Confirm password:		Password must be at least 8 characters.

Producing Messages from Anywhere

- It is also possible to add messages to the JSF context programmatically, from any code that is invoked in that context.
- Consider **Examples/Shopping/Step6**, which posts an informational message if, after valid inputs are provided, the control logic winds up merging quantities for an item that had already been ordered.
- See **src/cc/biz/web/ShoppingCart.java**, which watches for any purchased item whose key matches any of those already in the cart.
  - If any such items are found, it delegates to an **ErrorHandler**:

```
if (someItemsMerged)
  ErrorHandler.info
    ("One or more items were already ...");
```

Producing Messages from Anywhere

EXAMPLE

- The handler method “manually” adds a message to the current context – see **src/cc/jsf/ErrorHandler.java**.
  - One overload of the **info** method delegates to the other, passing **null** as the associated client ID; this is legal, and just means the message is “global:”

```
public static void info (String message)
{
  info (message, null);
}
```

  - The other overload adds a message to the context, taking the given string to be both summary and detail messages:

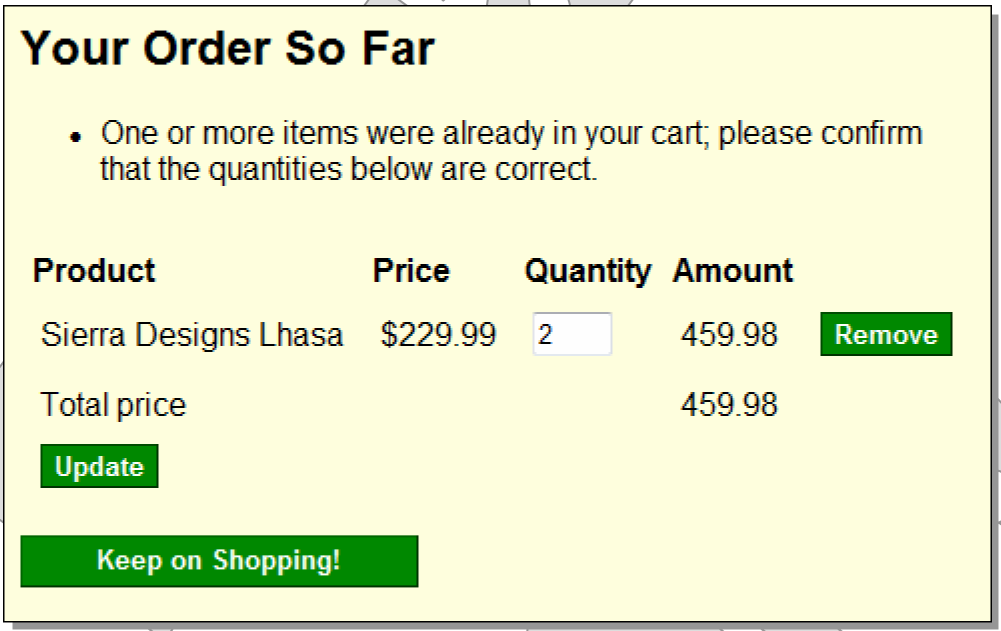
```
public static void info (String message, String ID)
{
  FacesContext.getCurrentInstance ()
    .addMessage (ID, new FacesMessage
      (FacesMessage.SEVERITY_INFO,
        message, message));
}
```

Producing Messages from Anywhere

EXAMPLE

- Adding messages to the context does not interrupt the JSF lifecycle in any way.
- But the next view presentation will be able to present those messages, using `<h:message(s)>` as usual.
- Build and test this version, and see that if you purchase some additional quantity of an item after adding it to the cart once, you'll get this message – which after all is just informational:

`http://localhost:8080/Shopping`



Your Order So Far

- One or more items were already in your cart; please confirm that the quantities below are correct.

Product	Price	Quantity	Amount	
Sierra Designs Lhasa	$229.99	2	459.98	Remove
Total price			459.98	

Validating Invoice Data

Suggested time: 30 minutes

In this lab you will add validation constraints to the invoice forms in the Billing application:

- Customer, invoice number, invoice date, and amount are all required fields.
- Amounts must not be negative numbers, nor zero.

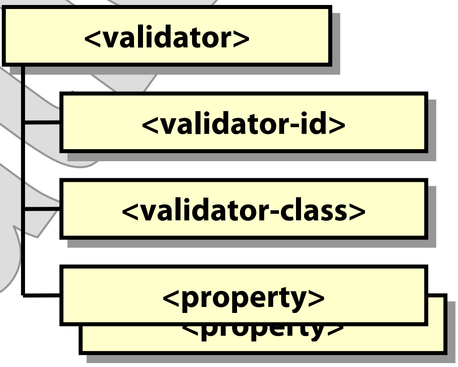
You will also set labels for all inputs, and enhance the page design to place field-specific error messages next to the input fields themselves, instead of relying on a summary at the top of the page as the starter code does.

Detailed instructions are found at the end of the chapter.

Custom Validators

- You can define your own validation logic in several ways – this is largely parallel to what we've seen for converters, except that there is no validator-for-class option:
  - Create a class that implements **Validator** and register it as a validator in the configuration file:

```
<validator>
  <validator-id>
    myValidator
  </validator-id>
  <validator-class>
    com.me.MyValidator
  </validator-class>
</validator>
```



  - Annotate your implementation as a **@FacesValidator**.

JSF 2.0

```
@FacesValidator
public class MyValidator
  implements Validator
```

  - Configure a **Validator** implementation as a managed bean.
  - Implement a method on any bean with the same signature as the **validate** method from **Validator**, but your own choice of method name. This can be a nice way to include validation logic on the backing bean itself, or on a related controller.

<f:validator> and the validator Attribute

- **Plug your validation logic into your view definitions using different techniques, depending on how the logic was defined:**
  - Attach an `<f:validator>` tag to any editable component, using the **validatorId** attribute to identify the validator you want.

```
<h:inputText value="#{bean.prop}" >
  <f:validator validatorId="myValidator" />
</h:inputText>
```

  - Use this same tag, but with a **binding** attribute, to identify a managed bean that implements **Validator**:

```
<h:inputText value="#{bean.prop}" >
  <f:validator binding="#{myValidatorBean}" />
</h:inputText>
```

  - Invoke a validation method using the **validator** attribute on the editable component to identify it:

```
<h:inputText
  value="#{bean.prop}"
  validator="#{bean.validateProp}"
/>
```

  - Using the reference implementation, you can also set **validator** to a validator ID – this is undocumented and non-standard:

```
  validator="myValidator"
```

  - Either way, one small issue with this last technique is that you can only do it **once per component**; you can’t mix and match validation methods the way you can with validator classes.

Conversion/Validation Lifecycle

EXAMPLE

- Partly as a simple example of a custom validator, and partly to explore the lifecycle implications of validation errors, we will observe a new version of the Lifecycle application.
- See **Examples/Lifecycle/Step8**, which has two new features:
  - A custom validator that traces calls to **validate**, and also can trigger a validation error, thus altering the request handling
  - Code in the existing **value-change listener** that invalidates user input later in the validations phase
- First, see **docroot/lifecycle.xhtml**, which now assigns a custom validator to the menu component:

```
<h:selectOneMenu
  value="#{bean.selection}"
  valueChangeListener=
    "#{bean.valueChangeListener}"
>
  <f:selectItems value="#{bean.selections}" />
  <f:valueChangeListener
    type="cc.jsf.ValueChangeListener" />
  <f:converter converterId="converterHook" />
  <f:validator validatorId="validatorHook" />
</h:selectOneMenu>
```

- The validator is declared in **docroot/faces-config.xml**:

```
<validator>
  <validator-id>validatorHook</validator-id>
  <validator-class>cc.jsf.ValidatorHook
    </validator-class>
</validator>
```

Conversion/Validation Lifecycle

EXAMPLE

- The implementation in **src/cc/jsf/ValidatorHook.java** is simple: it traces the call to **validate** and then succeeds for most values, but flunks one.

```
public void validate (FacesContext context,
    UIComponent component, Object value)
  throws ValidatorException
{
  System.out.print ("  Validator.validate() ... ");
  if (value.equals (Menu.Choice.UNSUBSCRIBE))
  {
    System.out.println ("FAILS.");
    throw new ValidatorException (new FacesMessage
      (FacesMessage.SEVERITY_ERROR,
        "No!", "Not allowed!"));
  }
  else
      System.out.println ("succeeds.");
}
```

- Build and test as usual. When you submit any value except **UNSUBSCRIBE**, we see the full lifecycle, with the validator called after the converter during the validations phase:

```
http://localhost:8080/Lifecycle

  ViewPhaseListener.before(PROCESS_VALIDATIONS 3)
    Converter.getAsObject()
    Validator.validate() ... succeeds.
    Menu.getSelections()
    Menu.getSelections()
    Menu.getSelection()
    Menu.valueChangeListener()
    ValueChangeListener.processValueChange()
  ViewPhaseListener.after(PROCESS_VALIDATIONS 3)
```

Conversion/Validation Lifecycle

EXAMPLE

- If you go back and submit **UNSUBSCRIBE**, you'll see that the lifecycle is shortened based on the validator's actions:

```
GlobalPhaseListener.before(PROCESS_VALIDATIONS 3)
  ViewPhaseListener.before(PROCESS_VALIDATIONS 3)
    Converter.getAsObject()
    Validator.validate() ... FAILS.
  ViewPhaseListener.after(PROCESS_VALIDATIONS 3)
GlobalPhaseListener.after(PROCESS_VALIDATIONS 3)

GlobalPhaseListener.before(RENDER_RESPONSE 6)
  ViewPhaseListener.before(RENDER_RESPONSE 6)
    Menu.getSelection()
    Menu.getSelections()
    Converter.getAsString()
    Converter.getAsString()
    Converter.getAsString()
    Converter.getAsString()
Jul 17, 2010 7:24:24 PM
com.sun.faces.renderkit.RenderKitUtils
renderUnhandledMessages
INFO: WARNING: FacesMessage(s) have been enqueued,
but may not have been displayed.
sourceId=form:j_idt3[severity=(ERROR 2),
summary=(No!), detail=(Not allowed!)]
  ViewPhaseListener.after(RENDER_RESPONSE 6)
GlobalPhaseListener.after(RENDER_RESPONSE 6)
```

  - Of course the expected page navigation is set aside as well.
  - Notice too a behavior of JSF that we've seen here and there during the course but not yet highlighted: it knows which error messages have been reported and which haven't, and it does the developer a kindness by dumping the unreported ones to the console.

Conversion/Validation Lifecycle

EXAMPLE

- Now, try **VOLUNTEER**: this also fails:

```
GlobalPhaseListener.before(PROCESS_VALIDATIONS 3)
  ViewPhaseListener.before(PROCESS_VALIDATIONS 3)
    Converter.getAsObject()
    Validator.validate() ... succeeds.
    Menu.getSelections()
    Menu.getSelections()
    Menu.getSelection()
    Menu.valueChangeListener()
    ValueChangeListener.processValueChange()
      Resetting valid flag on UIComponent ...
  ViewPhaseListener.after(PROCESS_VALIDATIONS 3)
GlobalPhaseListener.after(PROCESS_VALIDATIONS 3)

GlobalPhaseListener.before(UPDATE_MODEL_VALUES 4)
  ViewPhaseListener.before(UPDATE_MODEL_VALUES 4)
  ViewPhaseListener.after(UPDATE_MODEL_VALUES 4)
GlobalPhaseListener.after(UPDATE_MODEL_VALUES 4)

GlobalPhaseListener.before(RENDER_RESPONSE 6)
  ViewPhaseListener.before(RENDER_RESPONSE 6)
    Menu.getSelections()
    Converter.getAsString()
    Converter.getAsString()
    Converter.getAsString()
    Converter.getAsString()
Jul 17, 2010 7:26:36 PM
com.sun.faces.renderkit.RenderKitUtils
renderUnhandledMessages
INFO: WARNING: FacesMessage(s) have been enqueued,
but may not have been displayed.
sourceId=form:j_idt3[severity=(WARN 1),
summary=(Hello), detail=(Just kidding ..)]
  ViewPhaseListener.after(RENDER_RESPONSE 6)
GlobalPhaseListener.after(RENDER_RESPONSE 6)
```

Conversion/Validation Lifecycle

EXAMPLE

- This is the work of additional code in the pre-existing listener class. See **src/cc/jsf/ValueChangeListener.java**:

```
public void processValueChange (ValueChangeEvent ev)
{
  System.out.println
    ("  ValueChangeListener.processValueChange()");

  if (ev.getNewValue ().equals
      (Menu.Choice.VOLUNTEER))
  {
    System.out.println
      ("  Resetting valid flag on UIComponent...");
    ((EditableValueHolder) ev.getComponent ())
      .setValid (false);
    FacesContext.getCurrentInstance ().addMessage
      (ev.getComponent ().getClientId (),
        new FacesMessage
          (FacesMessage.SEVERITY_WARN,
            "Hello", "Just kidding .."));
  }
}
```

  - The processing of validators had already concluded by the time this code was called, so the shortening of the lifecycle didn’t occur until the next phase.
  - Still, no application of the model value occurred, so the difference isn’t substantial.

Validating Multiple Inputs

- **JSF makes validating single inputs pretty easy – even for more complex validation logic.**
- **It is weaker in its support for validation logic that requires multiple inputs:**
  - Making sure **passwords match**
  - Checking that start and end dates are in **chronological order**
  - **Requiring** one field only **if** another value is provided or is equal to some expected value
- **MVC frameworks tend to apply validation starting at the request scope and drilling down from there.**
  - This makes them better at multi-input validation, but less facile for single inputs.
- **You can "look outside" the scope of the single input you're given in any validator or validation method.**
  - Use the provided **UIComponent** and call navigation methods including **getParent**, **getChildren**, and **findComponent**.
  - Then derive other values from the form as needed and apply your multi-value constraints.
- **Though workable, this isn't a totally clean system.**
  - The eventual error message will be associated with the field to which the validator is attached, even though others are involved.
  - It's not obvious where to encode the ID(s) of the other input component(s) to minimize maintenance concerns.

Matching Passwords

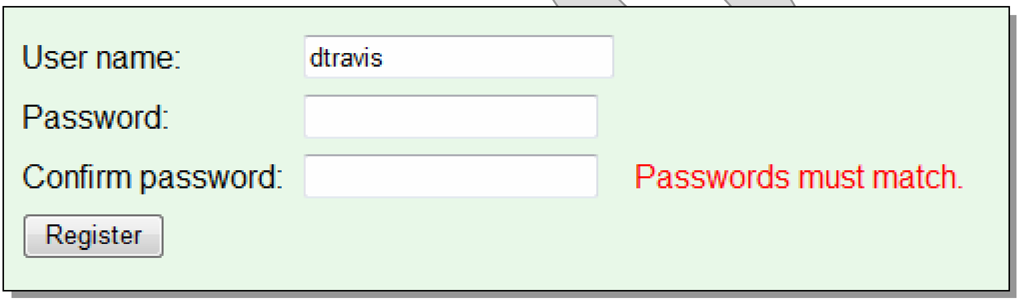
- **Examples/Passwords/Step4** has been enhanced with a validator that assures that the two passwords match.
- See **src/cc/jsf/PasswordValidator.java**:

```
public void validate (FacesContext context,
    UIComponent component, Object value)
  throws ValidatorException
{
  Object mainPassword = ((EditableValueHolder)
    component.findComponent (":form:password"))
      .getValue ();

  if (mainPassword != null &&
      !mainPassword.equals (value))
    throw new ValidatorException (new FacesMessage
      (FacesMessage.SEVERITY_ERROR,
        "Passwords must match.",
        "Passwords must match."));
}
```

  - Note that we hard-code the absolute ID of the first password field.

- Build and test, and see that, if all other validations succeed, but the passwords don't match, we see the associated error message:

`http://localhost:8080/Passwords`



User name:	dtravis	
Password:		
Confirm password:		Passwords must match.

Chronological Order

- **Examples/LandUse** also uses a custom validator for multi-field validation:
  - It assures that two date fields are in **chronological order**.
  - It assures that two others are in order and also separated by a mandatory **delay**.
- See **docroot/detail.xhtml**.
  - The proposed start date associates a specific method on a managed bean as a validator:

```
<td>Proposed start date:</td>
<td>
  <h:inputText
    id="useStart"
    label="start date"
    value="#{DB.selectedProposal.useStart.time}"
    required="true"
    validator=
      "#{dateValidator.startVsApplicationDate}"
  >
    <f:convertDateTime pattern="M/d/yy" />
  </h:inputText>
</td>
```

  - The end date does something similar:

```
validator="#{dateValidator.endVsStartDate}"
```

Chronological Order

EXAMPLE

- See **src/gov/usda/usfs/landuse/web/DateValidator.java** for the bean class.
  - **startVsApplicationDate** compares one date to another and insists on an "approval time" to boot:

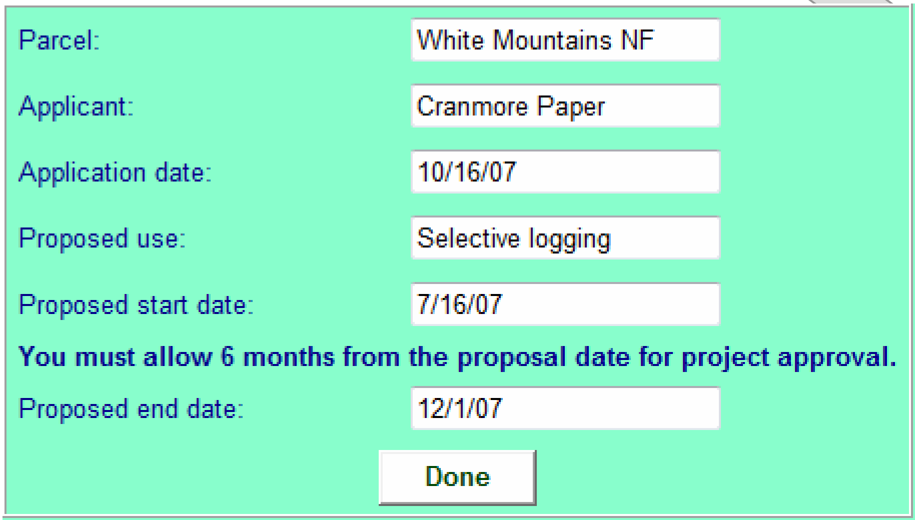
```
public void startVsApplicationDate
  (FacesContext context,
    UIComponent component, Object value)
  throws ValidatorException
{
  long startTime = ((Date) value).getTime ();
  long applicationTime =
    ((Date) ((ValueHolder) component
      .findComponent (":details:applicationDate"))
        .getValue ()).getTime ();

  if (applicationTime + APPROVAL_TIME > startTime)
    throw new ValidatorException (new FacesMessage
      (FacesMessage.SEVERITY_ERROR,
        "Starts too soon",
        "You must allow 6 months from the " +
          "proposal date for project approval."));
}
```

  - **endVsStartDate** does almost the same thing, with no lag time.

Chronological Order

- Test these constraints on any of the existing proposals:

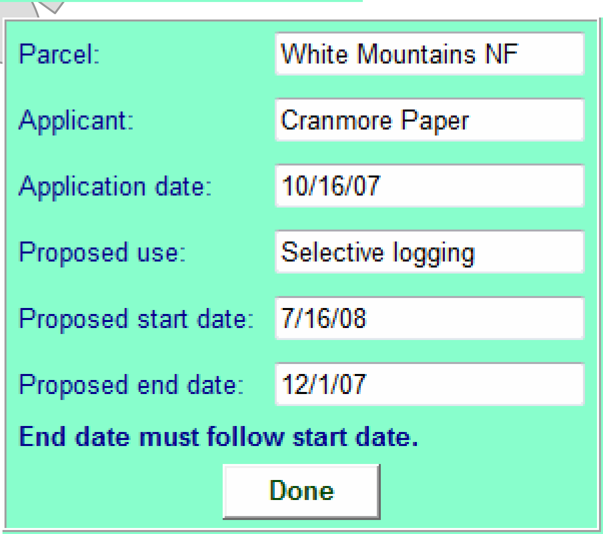
`http://localhost:8080/LandUse`



Parcel:	White Mountains NF
Applicant:	Cranmore Paper
Application date:	10/16/07
Proposed use:	Selective logging
Proposed start date:	7/16/07

You must allow 6 months from the proposal date for project approval.

Proposed end date:	12/1/07



Parcel:	White Mountains NF
Applicant:	Cranmore Paper
Application date:	10/16/07
Proposed use:	Selective logging
Proposed start date:	7/16/08
Proposed end date:	12/1/07

End date must follow start date.

Invoice and Payment Date

Suggested time: 45-60 minutes

In this lab you will add two custom validators to the Billing application. One is a generalization of the date-ordering validator we just saw for LandUse: it can be configured as to the client ID of the “other” date component, and it does a better job of presenting localizable error messages. This validator is complete, and you will just need to attach it to the payment-date component.

You will then build the second validator, which assures that a date is a business day – which, in a not-so-globally-robust fashion, we’ll define as being anything but Saturday or Sunday. You will then attach this validator to both date components, enforcing a rule that we don’t date either invoices or payments on weekends.

Detailed instructions are found at the end of the chapter.

JSF and "Bean Validation"

- A new validation standard enters the Java EE platform as of edition 6: this is known as **Bean Validation** or sometimes by its JSR number, 303.
- By this standard, any JavaBean can carry source-code annotations that declare validation constraints on its properties.
- These annotations can then be observed and enforced by a validation tool – at any time, in any tier of the application.
- So the advantage is that we can define validation constraints once, instead of having to write them out in different languages for different parts of a large application.
- JSF 2.0 supports Bean Validation automatically – <u>if</u> an implementation of JSR-303 is found on the class path.
    - In this case, whenever a backing property has any JSR-303 annotations, JSF will trigger the bean validator.
    - Error messages reported by the validator will be wrapped in **FacesMessage**s and added to the context, and the target component will be set to invalid.

Validation Annotations

- **Examples/JSR303** holds a simple Java SE application that validates values on two different JavaBeans.
- One of these is familiar – see **src/cc/math/Ellipsoid.java**:

```
@DecimalMin
(
  value=".0000001",
  message="Semi-axis A must be a positive number"
)
private double a = 1;
```

- The other, in **src/cc/web/PersonalInfo.java**, sets various constraints on its properties: rejecting **null** values and enforcing a regular expression, setting a value range, etc:

```
@NotNull
@Pattern
(
  regexp="([A-Za-z\\'\\-]+)( [A-Za-z\\'\\-]+)+",
  message="Must include at least ..."
)
private String name;

@Min
(
  value=18,
  message="Age must be at least 18"
)
@Max
(
  value=120,
  message="Age must be no greater than 120"
)
private int age;
```

Validation Annotations

- An application class creates a few instances of each type and applies the Bean Validator to them; we won't dig into this code as it's not directly relevant to JSF practice, since the JSF implementation will carry out this process for us.
- Build and test as follows:

```
ant
ant run

Ellipsoid "sphere":
  Validation succeeded.

Ellipsoid "twoD":
  Semi-axis B must be a positive number.

Ellipsoid "senseless":
  Semi-axis C must be a positive number.

Good PersonalInfo:
  Validation succeeded.

Bad PersonalInfo:
  Must include at least first and last name.
  Please keep reference to 40 characters or less.
  Invalid e-mail address.
  Invalid SSN.
  Age must be at least 18.
```

Using Existing Constraints

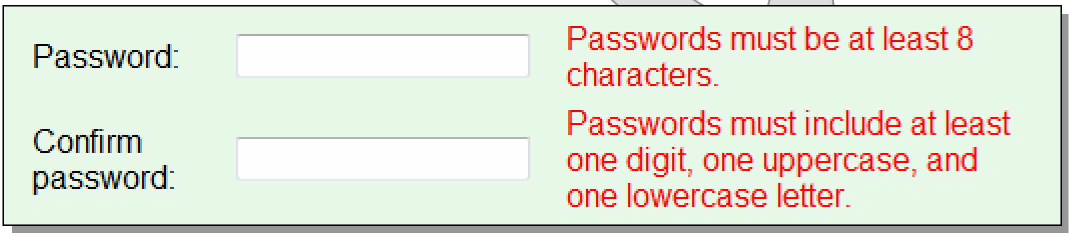
EXAMPLE

- **Examples/Passwords/Step5** uses constraint annotations instead of defining validators in the view.
- Here's the new backing bean – see **src/cc/jsf/Member.java**:

```
@Size(min=8, message=PASSWORD_LENGTH_MESSAGE)
@Pattern.List
({
    @Pattern(regexp=".*[A-Z].*",
             message=PASSWORD_FORM_MESSAGE),
    @Pattern(regexp=".*[a-z].*",
             message=PASSWORD_FORM_MESSAGE),
    @Pattern(regexp=".*[0-9].*",
             message=PASSWORD_FORM_MESSAGE)
})
private String password;
```

- In the view, the length and regular-expression validators have been removed.
  - We've kept the **required** attributes.
  - We've also kept the custom validator for password matching – this is more than we could manage with JSR-303.
- Build and test, and see the same basic logic, but with the new messages stemming from the source-code annotations:

`http://localhost:8080/Passwords`



Password:		Passwords must be at least 8 characters.
Confirm password:		Passwords must include at least one digit, one uppercase, and one lowercase letter.

Using <f:validateBean>

- The `<f:validateBean>` tag gives you some options regarding JSR-303 validation.
  - You can place this component as either an **ancestor or child** of one or more input components.

```
<h:inputText value="#{myBean.trickyProperty}" >
  <f:validateBean disabled="true" />
</h:inputText>

<f:validateBean validationGroups="#{groups}" >
  <h:inputText value="#{bean.prop1}" />
  <h:inputText value="#{bean.prop2}" />
  <h:inputText value="#{bean.prop3}" />
</f:validateBean>
```

  - You can set **validation groups** relevant to this form or to certain fields, thus filtering the possible validation constraints; this is going to be beyond our scope.
  - You can **disable** JSR-303 validation outright – again, for a field or for an entire form.

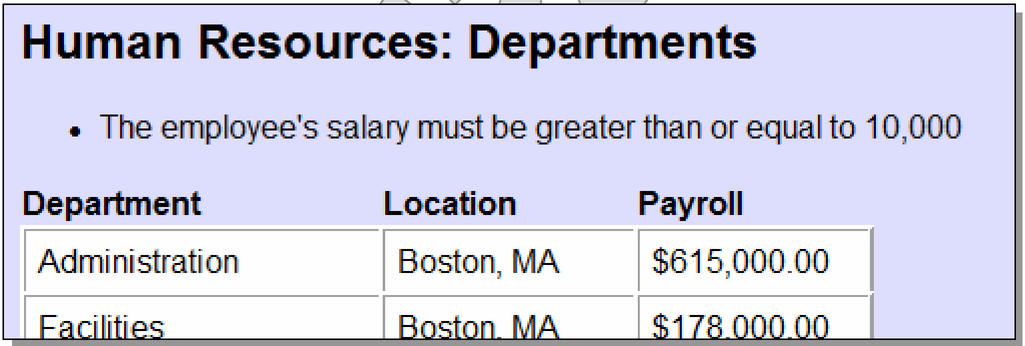
Disabling Bean Validation

- In **Demos/Disable**, we have the latest version of the HR application, and we'll experiment a bit with entering invalid salaries from the payroll-management page.

1. Build and test at the following URL:

```
http://localhost:8080/HR
```

2. Click the **Payroll** link.
3. Enter a salary of $1,000.00 for the first employee in the first department, and you'll see an error message:



Human Resources: Departments

- The employee's salary must be greater than or equal to 10,000

Department	Location	Payroll
Administration	Boston, MA	$615,000.00
Facilities	Boston, MA	$178,000.00

   – This is JSF observing the validation constraint on the backing bean – see **Examples/HR/JPA/src/cc/hr/entity/Employee.java**:

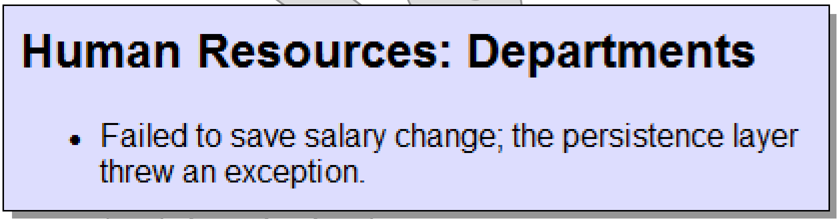
```
@Min(value = 10000,
     message = "The employee's salary must ...")
@Max(value = 9999999,
     message = "The employee's salary must ...")
private Long salary;
```

Disabling Bean Validation

4. Open **docroot/payroll.xhtml**, and (near the bottom of the file) disable JSR-303 validation for this field:

```
<h:inputText
  id="salary"
  value="#{employee.salary}"
  valueChangeListener="..."
>
  <f:convertNumber type="currency" />
  <f:validateBean disabled="true" />
</h:inputText>
```

5. Build and test again – what happens?



Human Resources: Departments

- Failed to save salary change; the persistence layer threw an exception.

- **You did indeed disable JSR-303 validation for the salary field – but only as it was being performed by the JSF runtime.**
- **There's another layer to this application, which is a system of JPA-2.0 façades and entities.**
- **JPA 2.0 also observes JSR-303 constraints!**
  - And of course this is the intended value of JSR-303: that we get validation and "re-validation" of values throughout an application, based on a central definition of validation constraints.
  - See the server console for <u>plenty</u> of detail on what JPA didn't like.

SUMMARY

- **Like the converter framework, JSF validation is simple to use and surprisingly powerful.**
- **Simple rules including required fields, string lengths, and value ranges can easily be declared as part of the view definition.**
  - Bear in mind that these simple rules account for the overwhelming majority of all input validation in web applications.
- **More complex logic can be plugged in by a handful of straightforward techniques.**
- **If there is a weak spot, it's multi-input validation.**
  - But a little extra logic to navigate the UI tree will bridge the gap between subject components.
- **Where JSR-303 validation constraints are available, JSF makes it easy to take full leverage from them.**