Berner Fachhochschule-Technik und Informatik

Advanced Web Technologies
6) JSF Validators and Converters

Dr. E. Benoist

Fall Semester 09-10

Table of Contents

- Motivations
  - Basic JSF Lifecycle
  - Why? / At what time?
  - Working Example: User Registration
- JSF Conversion
  - Default Converters
  - Format Patterns
  - Custom Converters
- Validators
  - Standard Validation Components
  - Application-level validation
  - Custom Validation Components
  - Validation methods in backing beans
- Conclusion

Convert and Validate Input?

- **Conversion and Validation are reusable**
  - Converting Numbers
  - Testing validity range
  - Testing the format of an input
  - Creating an object form a string
- **It is not part of the business logic**
  - Purpose: ensure values have been properly "sanitized" before updating model data.
  - We can concentrate on validation when required and then the input is considered OK.
- **Part of the input cycle**
  - It does not belong to navigation to return to the input page, when it is not good.
  - Error messages are automatically included in the page
  - Values are rewritten to prevent refilling everithing twice.

Baisc JSF Lifecycle

- Component tree retrieved

- Submitted form values stored in component
- Component values converted

- Component values validated

- Component values bound to backing bean properties

- Events handled
- Methods invoked
- Navigation evaluated

- Component values retrieved for rendering

Request

Restore view

Apply request values

Process validations

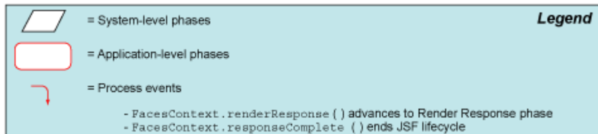
Update model values

Invoke application

Render response

Response

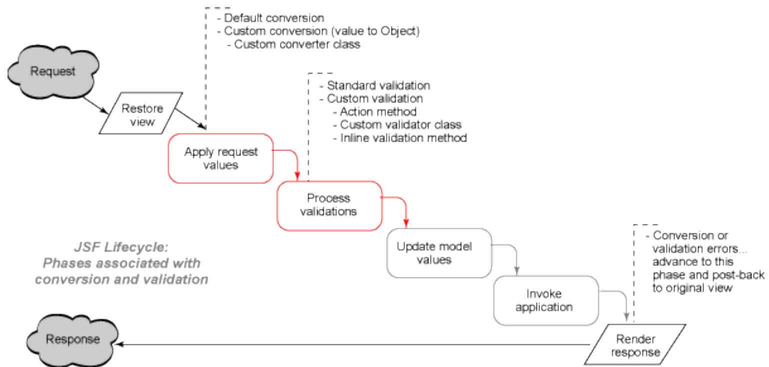
JavaServer Faces Basic Lifecycle



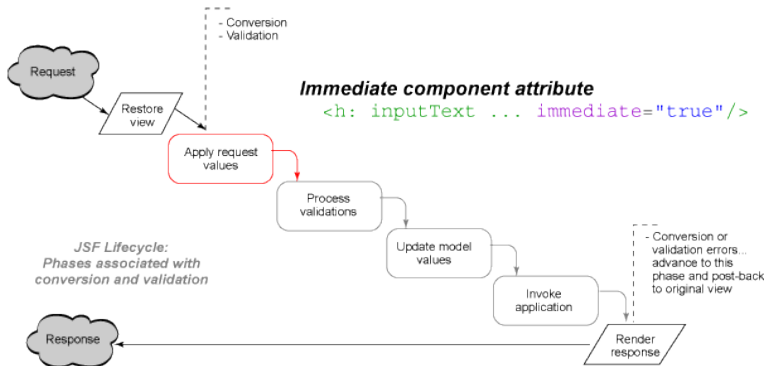
Conversion? Validations?

- **Conversion is the process of ensuring data is the right object or type.**
- **Two typical conversions**
  - string value is convertible to a `java.util.Date`
  - string value is convertible to a `Float`
- **Validation ensures that data contains the expected content**
- **Typical conversions**
  - `java.util.Date` is `MM/yyyy` format
  - `Float` is between 1.0 and 100.0

Details of the Lifecycle



When the immediate attribute is true



A working example: User Registration

- **This application will demonstrate**
  - Usage of standard JSF converters for converting form field data
  - Usage of standard JSF validation components for validating form field data
  - How to write custom converters and validator
  - How to register custom converters and validators in the `faces-config.xml` file
  - How to customize default error messages
- **Application utilizes three JSP pages**
  - `index.jsp` redirects the user to `UserRegistration.jsp`
  - `UserRegistration.jsp` contains the application's form fields
  - `results.jsp` notifies the application that the user was registered

JSF Conversion

- **JSF supplies many standard data converters.**
- **You can also plug in your own custom converter by implementing the** `Converter` **interface**
- **List of given** `Converters`

```
javax.faces.BigDecimal   javax.faces.convert.BigDecimalConverter
javax.faces.BigInteger   javax.faces.convert.BigIntegerConverter
javax.faces.Boolean      javax.faces.convert.BooleanConverter
javax.faces.Byte         javax.faces.convert.ByteConverter
javax.faces.Character    javax.faces.convert.CharacterConverter
javax.faces.DateTime     javax.faces.convert.DateTimeConverter
javax.faces.Double       javax.faces.convert.DoubleConverter
javax.faces.Float        javax.faces.convert.FloatConverter
```

Demonstration of the Default Converters

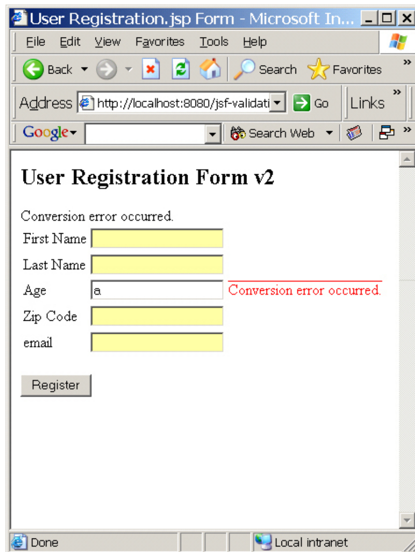
- **JSF tag**

```
<!-- UserRegistration.jsp -->
<h:inputText id="age" value="#{UserRegistration.user.age}"/>
```

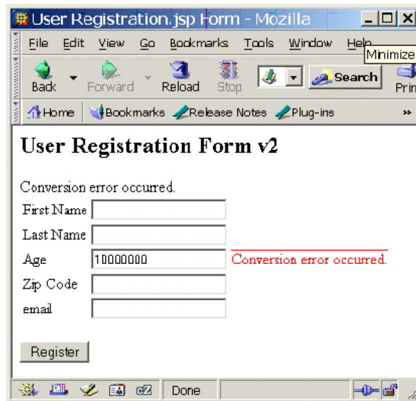
- `UserRegistration.user.age` **represents a value-binding property of type** `int`**.**
  - JSF uses standard converter
  - It is also used for BigInteger and BigDecimal
- **You can also increase granularity using a given converter**

```
<!-- UserRegistration.jsp -->
<h:inputText id="age" value="#{UserRegistration.user.age}">
    <f:converter id="javax.faces.Short"/>
</h:inputText>
```

Error authomatically generated



Error, value is too large



Choosing a Date Format Pattern

- **For dates, you must specify the conversion tag** `<f:convertDateTime/>`
- **Example:**

```
<!-- UserRegistration.jsp -->
<h:inputText id="birthDate"
                  value="#{UserRegistration.user.birthDate}">
    <f:convertDateTime pattern="MM/yyyy"/>
</h:inputText>
```

Additional Patterns

- **JSF provides a special converter for dealing with numbers such as percentages or currency**
  - It deals with grouping (such as commas), number of decimal digits, currency symbols, and such.
- **Example**

```
<!-- UserRegistration.jsp -->
<h:inputText id="salary"
             value="#{UserRegistration.user.salary}">
   <f:convertNumber maxFractionDigits="2"
        groupingUsed="true"
        currencySymbol="$"
        maxIntegerDigits="7"
        type="currency"/>
</h:inputText>
```

Incorrectly formated currency data

User Registration.jsp Form - Microsoft Inter...

File Edit View Favorites Tools Help

Address http://localhost:8080/jsf-validation/

User Registration Form v2

Conversion error occurred

First Name	Richard	
Last Name	Hightower	
Age	34	
Zip Code	85748	
email	rick@arc-mind.com	
Salary	80.0000	Conversion error occurred
Birth Date	05/1970	

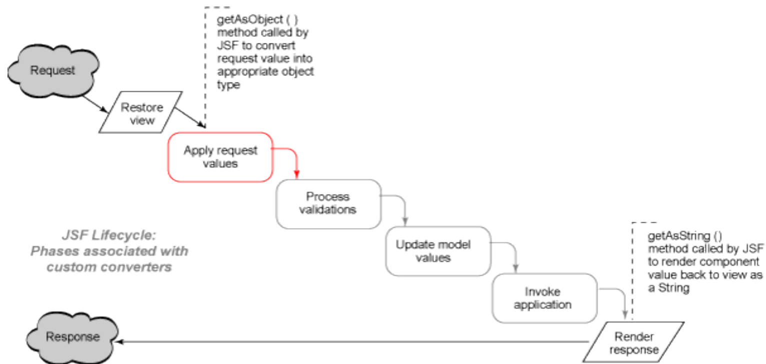
Register

Done Local intranet

Custom Converters

- **Necessary to convert field data into an application-specific value object**
  - String to PhoneNumber object (PhoneNumber.areaCode, PhoneNumber.prefix)
  - String to Name object (Name.first, Name.last)
- **You must follow the following steps**
  1. Implement the `Converter` interface (a.k.a. `javax.faxes.convert.Converter`).
  2. Implement the `getAsObject` method, which converts a field (string) into an object (for example, `PhoneNumber`).
  3. Implement the `getAsString` method, which converts an object (for example, `PhoneNumber`) into a string.
  4. Register your custom converter in the Faces context.
  5. Insert the converter into your JSPs with the `<f:converter/>` tag.

Custom Methods in JSF Lifecycle



Creating a custom converter

- **Step 1: Implement the `Converter` interface**

```
import javax.faces.convert.Converter;
import org.apache.commons.lang.StringUtils;
...

public class PhoneConverter implements Converter {
        ...

}
```

Step 2: getAsObject method

```
public class PhoneConverter implements Converter {
    ...
    public Object getAsObject(FacesContext context,
            UIComponent component, String value) {
        if (StringUtils.isEmpty(value)){ return null;}
        PhoneNumber phone = new PhoneNumber();
        String [] phoneComps = StringUtils.split(value," ,()-");
        String countryCode = phoneComps[0];
        phone.setCountryCode(countryCode);
        if ("1".equals(countryCode)){
            String areaCode = phoneComps[1];
            String prefix = phoneComps[2];
            String number = phoneComps[3];
            phone.setAreaCode(areaCode);
            phone.setPrefix(prefix);
            phone.setNumber(number);
        }else{ phone.setNumber(value);}
        return phone;
    }}
```

Custom Converter (Cont.)

- **Step 3: Implement the `getAsString` method**

```
public class PhoneConverter implements Converter {
    ...
    public String getAsString(FacesContext context,
                              UIComponent component, Object value) {
        return value.toString();
    }
}
public class PhoneNumber implements Serializable {
    ...
    public String toString(){
        if (countryCode.equals("1")){
            return countryCode + "␣" + areaCode
                                + "␣" + prefix + "␣" + number;
        }else{
            return number;
        }
    }
}
```

Custom Converter (Cont.)

- **Step 4: Register custom converter with faces context**
  Step 4 can be executed in one of two ways.
- Register the PhoneConverter class with an id

```
<converter>
  <converter-id>arcmind.PhoneConverter</converter-id>
  <converter-class>com.arcmind.converters.PhoneConverter</converter-class>
</converter>
```

- register the PhoneConverter class to handle all PhoneNumber objects automatically,

```
<converter>
  <converter-for-class>com.arcmind.value.PhoneNumber</converter-for-class>
  <converter-class>com.arcmind.converters.PhoneConverter</converter-class>
</converter>
```

Custom Converter (Cont.)

- **Step 5: Use the converter tag in your JSPs?**
- If converter registered with id `arcmind.PhoneConverter`

```
<h:inputText id="phone"
                value="#{UserRegistration.user.phone}">
  <f:converter converterId="arcmind.PhoneConverter" />
</h:inputText>
```

- If you chose to register the PhoneConverter class to handle all PhoneNumber objects automatically then you won't need to use the `<f:converter/>` tag in your JSPs.

```
<h:inputText id="phone"
                value="#{UserRegistration.user.phone}">
     [Look, no converter!]
</h:inputText>
```

Conversions

- **JSF supplied Data converters**
  - `<f:converter/>`
  - `<f:convertDateTime/>`
  - `<f:convertNumber/>`
- **Custom Data converters**
  - `<f:converter/>`

User Registration.jsp Fo...

File Edit View Go Bookmarks Tools

Back Forward Reload Stop

Home Bookmarks Release Notes

User Registration Form v2

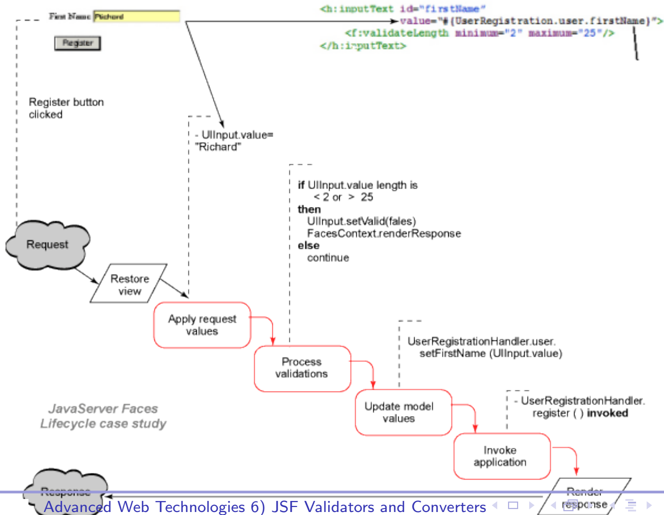
First Name	Richard
Last Name	Hightower
Age	34
Zip Code	85748
email	rick@rick.com
Salary	$0.00
Birth Date	05/1970
Phone	1 555 555 5555

Register

JSF Validation

- **Esures that input fullfil some requirements**
  - `java.util.Date` is `MM/yyyy` format
  - `Float` is between 1.0 and 100.0
- **There are four forms of validation within JSF:**
  - Built-in validation components
  - Application-level validation
  - Custom validation components (which implement the Validator interface)
  - Validation methods in backing beans (inline)

The JSF validation lifecycle and components



Standard Validation Components

- `DoubleRangeValidator`: **Component's local value must be numeric type; must be in range specified by minimum and/or maximum values.**
- `LongRangeValidator`: **Component's local value must be numeric type and convertible to long; must be in range specified by minimum and/or maximum values.**
- `LengthValidator`: **Type must be string; length must be in range specified by minimum and/or maximum values.**

Standard validation

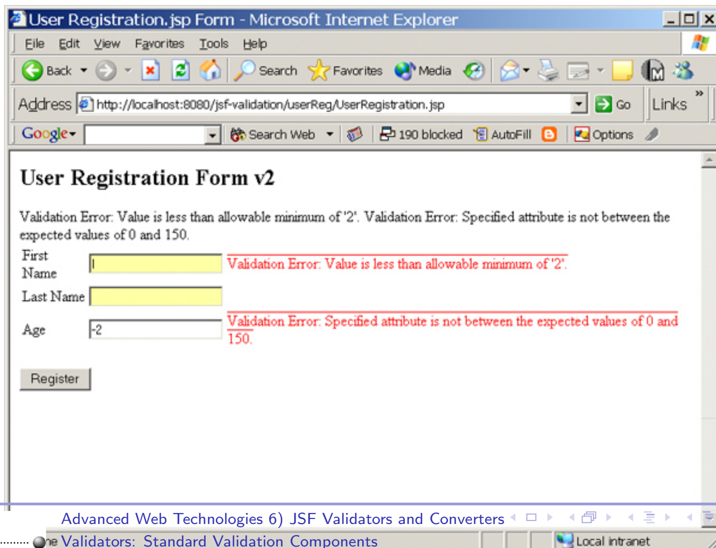
- **Age has to be between 0 and 150**

```
<h:inputText id="age" value="#{UserRegistration.user.age}">
            <f:validateLongRange maximum="150"
                                 minimum="0"/>
</h:inputText>
```

- **Length restrictions on the first-name field.**

```
<h:inputText id="firstName"
             value="#{UserRegistration.user.firstName}">
    <f:validateLength minimum="2"
                      maximum="25" />
</h:inputText>
```

Standard Validation Error Messages



Application-level Validation

- **In concept, application-level validation is really business logic validation.**
  - JSF separates form- and/or field-level validation from business-logic validation.
  - application-level validation entails adding additional code to the backing bean methods that use the model to qualify the data already bound to your model.
  - In the case of a shopping cart, form-level validation may validate whether a quantity entered is valid, but you would need business-logic validation to check whether the user had exceeded his or her credit limit.

Application-level Validation (Cont.)

- **Example**
  - The user clicks **Register** button which executes the `register()` method.
  - We could add validation code to the `register()` method to determine whether the first-name field is blank or null.
  - In cases where the field was null, we could also add a message to the FacesContext directing the associated component to return navigation to the current page.
  - (This example is not a good example of business logic validation)

Application-level validation

- Validation steps
  - Obtain the FacesContext
  - Validate the field (check to see if it is empty)
  - If validation error: Create a FacesMessage
    - Populate severity as error
    - Populate summary
    - Populate detail
  - Add facesMessage to FacesContext

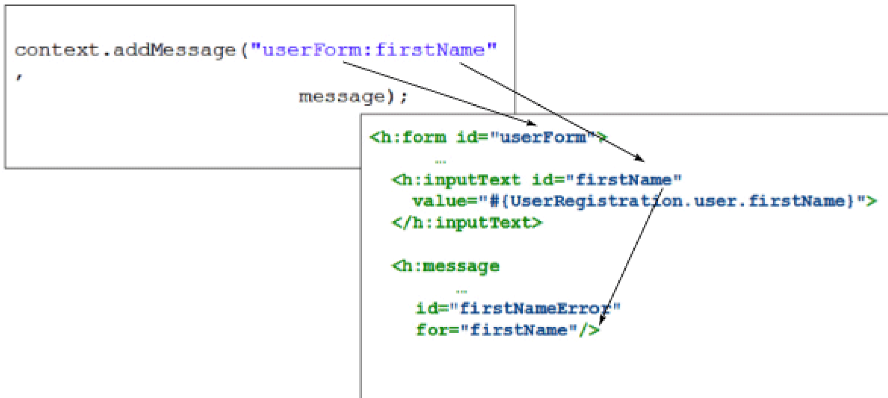
```
public String register() {
  FacesContext context =
          FacesContext.getCurrentInstance();

  if(StringUtils.isEmpty(user.getFirstName())){
    FacesMessage message = new FacesMessage();
    message.setSeverity(
            FacesMessage.SEVERITY_ERROR);
    message.setSummary("First name is blank");
    message.setDetail("First name is blank…");

    context.addMessage("userForm:firstName",
                       message);
    return "error";
  }
  return "success";
```

Validation Message

- Message is added as $ {formId}:${fieldId}
- <h:message associated with fieldId
- (use <h:messages to display all messages)



Pros and cons of application-level validation

- **The advantages of application-level validation are as follows:**
  - Easy to implement
  - No need for a separate class (custom validator)
  - No need for page author to specify validator
- **The disadvantages of application-level validation are as follows:**
  - Occurs after other forms of validation (standard, custom)
  - Validation logic limited to backing bean method, resulting in limited re-use
  - Can be difficult to manage in large applications and/or team environment
- **Ultimately, application-level validation should be used only for circumstances requiring business-logic validation.**

Custom Validation Components

- **The steps to create a custom validator are as follows; we'll go over them one by one:**
  - Create a class that implements the `Validator` interface (`javax.faces.validator.Validator`).
  - Implement the `validate` method.
  - Register your custom validator in the `faces-config.xml` file.
  - Use the `<f:validator/>` tag in your JSPs.

Custom Validation (Cont.)

- **Step 1: Implement the Validator interface**

```
import javax.faces.validator.Validator;
import javax.faces.validator.ValidatorException;
...
public class ZipCodeValidator implements Validator{
  private boolean plus4Required;
  private boolean plus4Optional;
  /** Accepts zip codes like 85710 */
  private static final String ZIP_REGEX = "[0-9]{5}";
  /** Accepts zip code plus 4 extensions like "-1119" or " 1119" */
  private static final String PLUS4_REQ_REGEX = "[ |-]{1}[0-9]{4}";
  /** Optionally accepts a plus 4 */
  private static final String PLUS4_OPT_REGEX = "([ |-]{1}[0-9]{4})?";
  ...
}
```

Custom Validation (Cont.)

- **Step 2: Implement the validate method**

```
public void validate(FacesContext context, UIComponent component,
                     Object value) throws ValidatorException {
  Pattern mask = null;
  initProps(component);
  if (plus4Required){
    mask = Pattern.compile(ZIP_REGEX + PLUS4_REQ_REGEX);
  } else if (plus4Optional){
    mask = Pattern.compile(ZIP_REGEX + PLUS4_OPT_REGEX);
  } else if (plus4Required && plus4Optional){
    throw new IllegalStateException("Plus 4 Error ...");
  }
  else {
    mask = Pattern.compile(ZIP_REGEX);
  }
```

Custom Validation (Cont.)

```
    String zipField = (String)value;
    Matcher matcher = mask.matcher(zipField);
    if (!matcher.matches()){
      FacesMessage message = new FacesMessage();
      message.setDetail("Zip code not valid");
      message.setSummary("Zip code not valid");
      message.setSeverity(FacesMessage.SEVERITY_ERROR);
      throw new ValidatorException(message);
    }
}
```

Custom Validation (Cont.)

- **Step 3: Register your custom validator with the FacesContext**

```
<validator>
   <validator-id>arcmind.zipCodeValidator</validator-id>
   <validator-class>
         com.arcmind.jsfquickstart.validation.ZipCodeValidator
   </validator-class>
</validator>
```

Custom Validation (Cont.)

- **Step 4: Use the `<f:validator/>` tag in your JSPs**

```
<h:inputText id="zipCode" value="#{UserRegistration.user.zipCode}">
    <f:validator validatorId="armind.zipCodeValidator"/>
    <f:attribute name="plus4Optional" value="true"/>
</h:inputText>
```

To read the `plus4Optional` attribute for the `zipCode inputText` component, do the following:

```
private void initProps(UIComponent component) {
  Boolean optional = Boolean.valueOf((String) component.getAttributes().
                                      get("plus4Optional"));
  Boolean required = Boolean.valueOf((String) component.getAttributes().
                                      get("plus4Required"));
  plus4Optional = optional==null ? plus4Optional :
                                          optional.booleanValue();
  plus4Required = required==null ? plus4Optional :
                                          required.booleanValue();
}
```

Validation methods in backing beans

- **you can simply implement custom validation in a backing bean method,**

  [SomeBackingBean.java]

```
public void validateEmail(FacesContext context,
                          UIComponent toValidate,
                          Object value) {
    String email = (String) value;

    if (email.indexOf('@') == -1) {
        ((UIInput)toValidate).setValid(false);

        FacesMessage message = new FacesMessage("Invalid Email");
        context.addMessage(toValidate.getClientId(context), message);
    }
}
```

Validation methods (Cont.)

- **The method would then be used in the JSF tag via the** `validator` **attribute as shown here:**

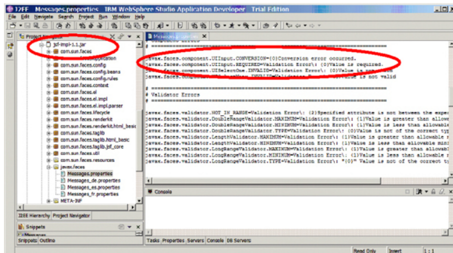
```
<h:inputText id="email"
             value="#{UserRegistration.user.email}"
             validator="#{UserRegistration.validateEmail}"
             required="true">
</h:inputText>
```

Default validation

- **Notice the required attribute of the email tag of the previous slide**
  - Utilizing the required attribute is a form of default validation.
  - If the attribute is true then the corresponding component must have a value.

Custom messages

- it's possible to change the default messages supplied by JSF by creating your own message resource bundle.
- Contained in the jsf-impl.jar (or similar) is a message.properties file that contains the default messages shown

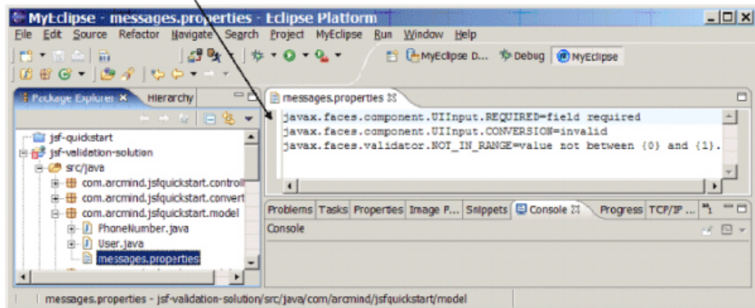


Switching out the message resource bundles

- Modify faces-config.xml: add message resource bundle

- Add replacement messages

```
[faces-config.xml]
<application>
    <locale-config>
        <default-locale>en</default-locale>
    </locale-config>
    <message-bundle>
        com.arcmind.jsfquickstart.model.messages
    </message-bundle>
</application>
```



Conclusion

- **conversion and validation don't necessarily work well together.**
  - Conversion converts strings into objects, whereas most of the standard validators work on strings.
  - Therefore, you must execute caution when using custom convertors and validators together.
  - For instance, our PhoneNumber object would not work with a length validator.
  - In this case, you would either have to write a custom validator, as well,
  - or simply include any special validation logic in the custom converter.
  - We prefere the later option because it allows us to simply associate a custom converter (with built-in validation logic) with a specific object type and have JSF handle that object type.

References

- http://www-128.ibm.com/developerworks/java/library/j-jsf3/