Java Server Faces (JSF)

Table of Contents

Chapter 7

Conversion and Validation

Overview

Presentation View

Strings

Model View

Java Data Types

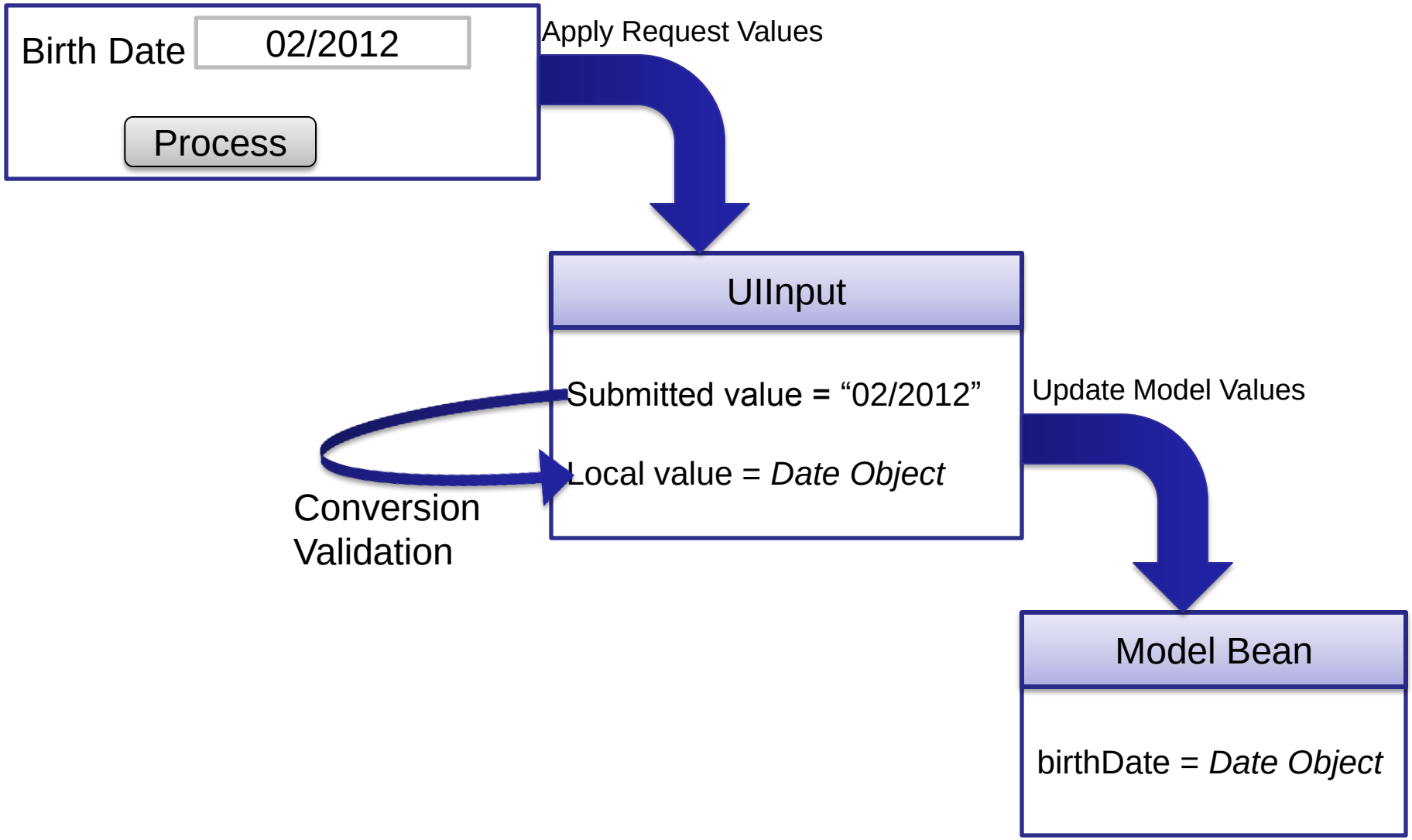
Browser

- <input type="text" name="birthdate" value="#{bean.birthDate}">

Web Application

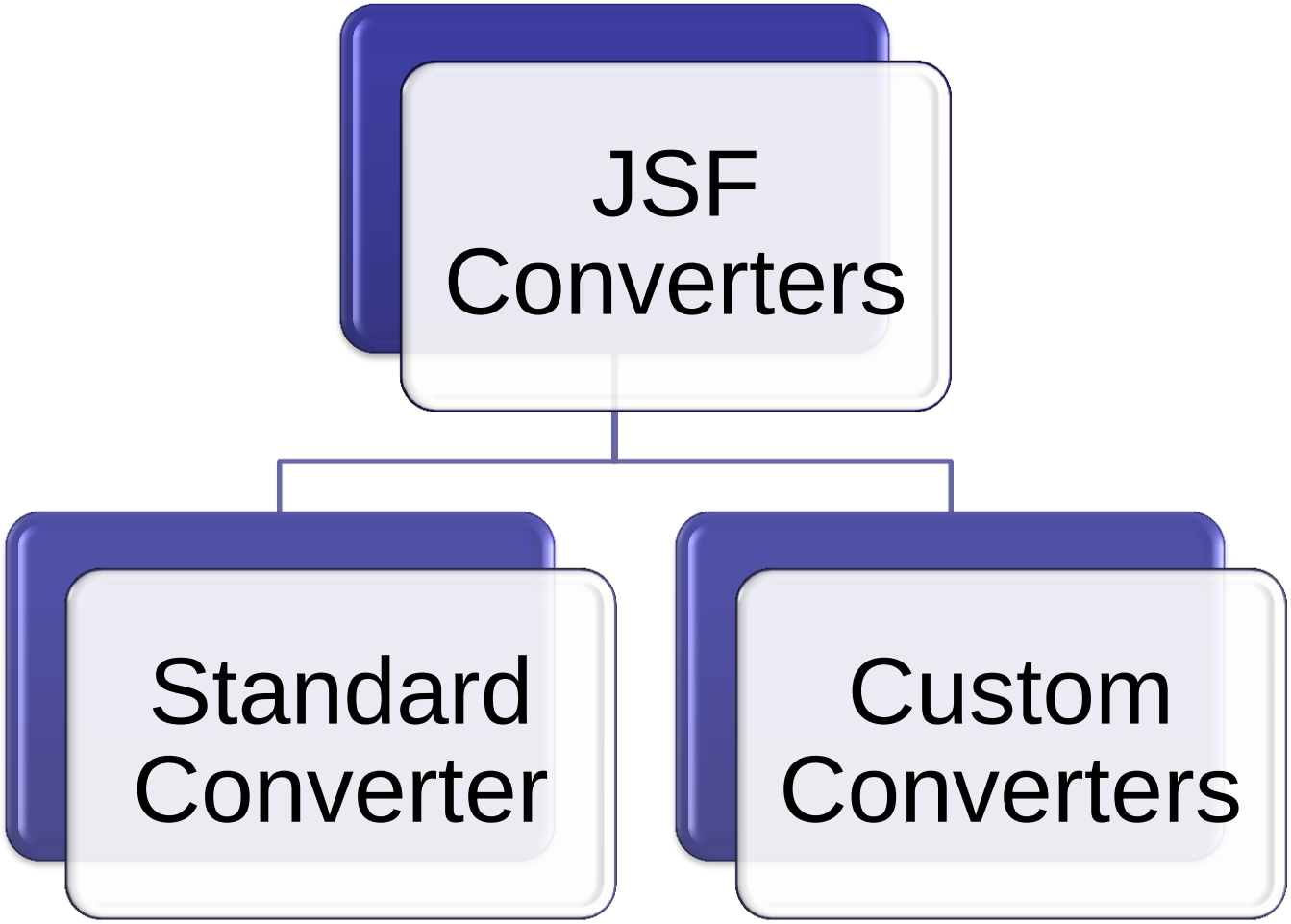
- java.util.Date birthDate;

Overview Cont.



Converters

Types of Converters



Standard Converters

- javax.faces.DateTime
- javax.faces.Number
- javax.faces.Boolean
- javax.faces.Byte
- javax.faces.Character
- javax.faces.Double
- javax.faces.Float
- javax.faces.Integer
- javax.faces.Long
- javax.faces.Short
- javax.faces.BigDecimal
- javax.faces.BigInteger

Standard Converters Cont'd

- **Two Ways to Use Standard Converters**

Standard Converter Tag

```
<h:outputText value="#{payment.date}">
     <f:convertDateTime/>
</h:outputText>
```

Inner Converter Tag

```
<h:outputText value="#{payment.date}">
<f:converter converterId="javax.faces.DateTime"/>
</h:outputText>
```

Converter Attribute

```
<h:outputText value="#{payment.date}"
converter="javax.faces.DateTime"/>
```

Standard Converters Cont'd

- **<f:convertNumber> Tag**
  - This tag used to convert numbers to different representations like currency, percentage etc.

```
<h:outputText value="#{payment.amount}">
<f:convertNumber type="currency"/>
</h:outputText>
```

- **<f:convertDateTime> Tag**
  - This Tag is used to Convert the String to a Date/Time object and to Display it in Different Formats.

```
<h:inputText value="#{payment.date}">
<f:convertDateTime pattern="MM/yyyy"/>
</h:inputText>
```

<f:convertNumber> Attributes

Attribute	Type	Value
type	String	number (default), currency, or percent
pattern	String	Formatting pattern, as defined in java.text.DecimalFormat
maxFractionDigits	int	Maximum number of digits in the fractional part
minFractionDigits	int	Minimum number of digits in the fractional part
maxIntegerDigits	int	Maximum number of digits in the integer part
minIntegerDigits	int	Minimum number of digits in the integer part
integerOnly	boolean	True if only the integer part is parsed (default: false)
groupingUsed	boolean	True if grouping separators are used (default: true)
locale	java.util.Locale	Locale whose preferences are to be used for parsing and formatting
currencyCode	String	ISO 4217 currency code to use when converting currency values
currencySymbol	String	Currency symbol to use when converting currency values

<f:convertDateTime> Attributes

Attribute	Type	Value
type	String	date (default), time, or both
dateStyle	String	default, short, medium, long, or full
timeStyle	String	default, short, medium, long, or full
pattern	String	Formatting pattern, as defined in java.text.SimpleDateFormat
locale	java.util.Locale	Locale whose preferences are to be used for parsing and formatting
timeZone	java.util.TimeZone	Time zone to use for parsing and formatting

Display Conversion Error Messages

- You can provide a custom converter error message for a component. Set the converterMessage attribute of the component whose value is being converted.

Example

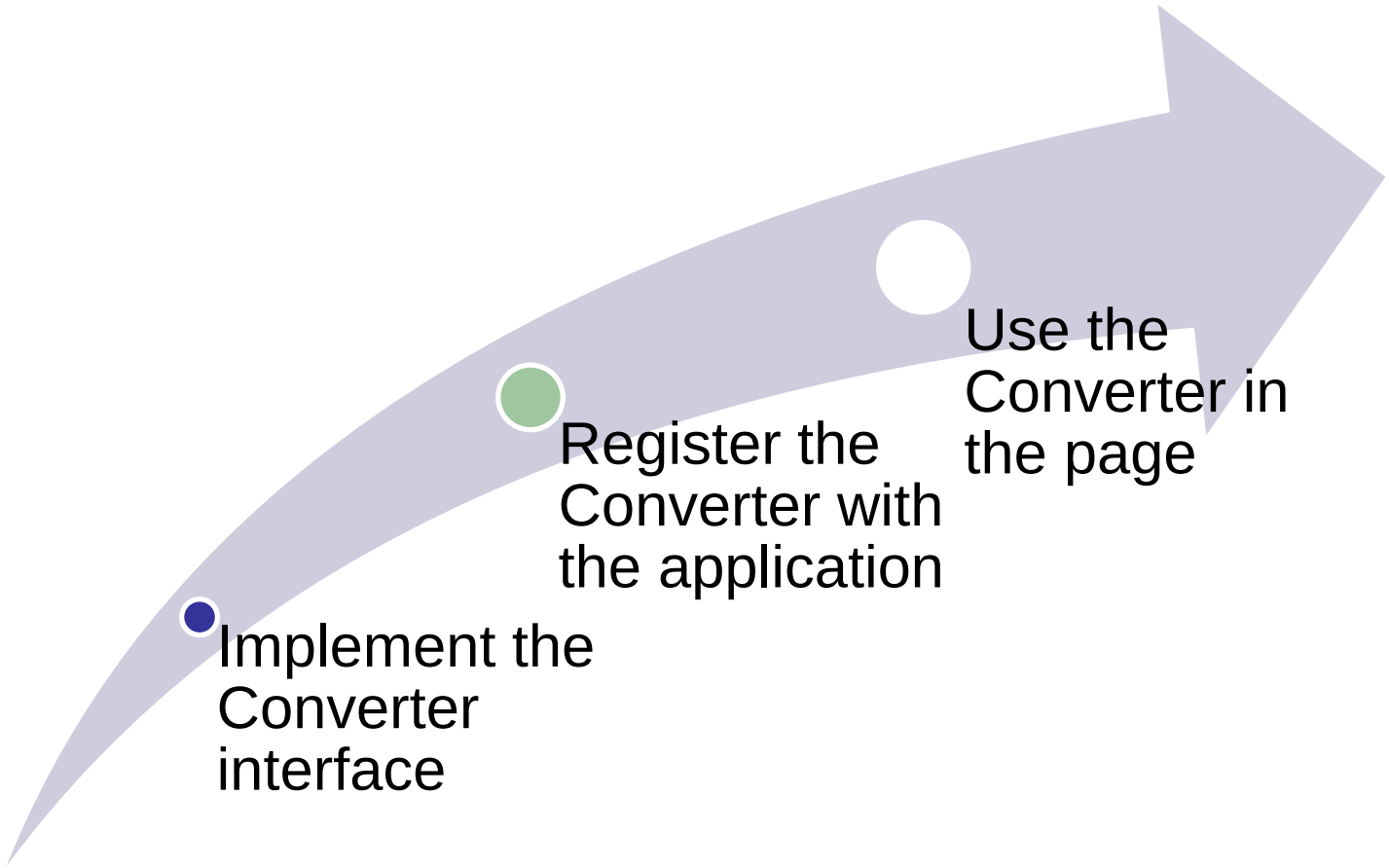
- `<h:inputText ... converterMessage="Not a valid number."/>`

Custom Converters

- **Why we need custom converters ?**
  - If you want to convert a component's data to a type other than a standard type
  - If you want to convert the format of the data
- **<u>CreditCard Converter:</u>**
  - accepts a Credit Card Number of type String and blanks and "-".
  - It also formats the CreditCardNumber such a blank space separates every four characters.

Custom Converters Cont.

- **Steps for Creating/Using Custom Converter:**



Custom Converters Cont.

- **Step1: Implement Converter Interface**
  - Define how to convert data both ways between the two views of the data:
  - Presentation view to Model view

    Object getAsObject(FacesContext, UIComponent, String)
  - Model view to Presentation view

    String getAsString(FacesContext, UIComponent, Object)

Custom Converters Cont.

- **Step2: Register Converter**
  - You associate the ID with the converter in one of two ways.

@FacesConverter annotation

```
@FacesConverter("com.corejsf.Card")
public class CreditCardConverter implements
Converter
```

Add entry to faces-config.xml

```
<converter>
<converter-id>com.corejsf.Card</converter-id>
<converter-class>
com.corejsf.CreditCardConverter
</converter-class>
</converter>
```

Custom Converters Cont.

- **Step3: Use the Converter in the Page**
  - Now we can use the **f:converter** tag and specify the converter ID:

Example

```
<h:inputText value="#{payment.card}">
<f:converter converterId="com.corejsf.Card"/>
</h:inputText>
```

  - Also we can say :

Example

```
<h:inputText value="#{payment.card}"
converter="Ccom.corejsf.Card" />
```

Register Default Converter to Class

- If you are confident that your converter is appropriate for all conversions between String and your class objects, then you can register it as the default converter for the your class.

Using Annotation

```
•@FacesConverter(forClass=CreditCard.class)
```

Using faces-config.xml

```
•<converter>
•<converter-for-class>com.corejsf.CreditCard</converter-
 for-class>
•<converter-
 class>com.corejsf.CreditCardConverter</converter-class>
•</converter>
```

- Now you do not have to mention the converter any longer.

Reporting Conversion Errors

- When a converter detects an error, it should throw a ConverterException.

Example

```
• if (foundInvalidCharacter) {
  FacesMessage message = new FacesMessage( "Conversion
  error occurred. ", "Invalid card number. ");
• message.setSeverity(FacesMessage.SEVERITY_ERROR);
• throw new ConverterException(message);
  }
```

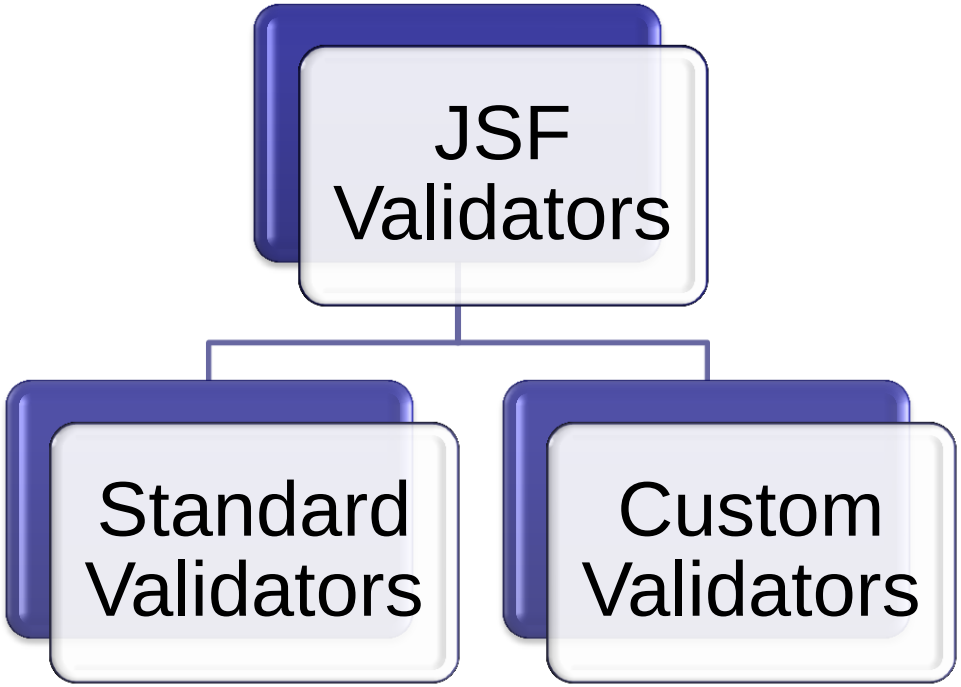
Validators

Validation

- **When to Use Validators?**

  Validation can only be performed on *UIInput* components or components whose classes extend *UIInput.*

- **Types of Validators:**



Standard Validators

JSP Tag	Validator Class	Attributes	Validates
f:validateDoubleRange	DoubleRangeValidator	Minmum Maximum	A double value within an optional range
f:validateLongRange	LongRangeValidator	Minmum Maximum	A long value within an optional range
f:validateLength	LengthValidator	Minmum Maximum	A String with a minimum and maximum number of characters
f:validateRequired	RequiredValidator		The presence of a value
f:validateRegex	RegexValidator	pattern	A String against a regular expression

Standard Validators Cont.

- Examples

<f:validateLongRange>

```
<h:inputText id="amount" value="#{payment.amount}">
<f:validateLongRange minimum="10" maximum="10000"/>
</h:inputText>
```

<f:validateRequired>

```
<h:inputText id="date" value="#{payment.date}">
<f:validateRequired/>
</h:inputText>
```

<f:validateLength minimum="13"/>

```
<h:inputText id="card" value="#{payment.card}">
<f:validateLength minimum="13"/>
</h:inputText>
```

Standard Validators Cont.

- An alternate syntax for attaching a validator to a component is to use the f:validator tag.

Example

```
<h:inputText id="card" value="#{payment.card}">
<f:validator
validatorId="javax.faces.validator.LengthValidator">
<f:attribute name="minimum" value="13"/> </f:validator>
</h:inputText>
```

Display Error Messages

- You can provide a custom validator error message for a component.

Example

- `<h:inputText ...`

  `validatorMessage="invalid length ."/>`

Display Validation Error Messages

- you can supply a custom message for a component by setting the requiredMessage or validatorMessage attribute

Example

```
<h:inputText id="card"
value="#{payment.card}" required="true”
requiredMessage="#{msgs.cardRequired}”
validatorMessage="#{msgs.cardInvalid}">
<f:validateLength minimum="13"/>
</h:inputText>
```

Bypassing Validation

- Validation errors (as well as conversion errors) force a redisplay of the current page.
- This behavior can be problematic with certain navigation actions.
- For example, you add a Cancel button to a page that contains required fields.

Bypassing validation Cont.

*User Name

*Password

Login Cancel

Please Fill The Required Fields

*User Name

*Password

Login Cancel

Bypassing validation Cont.

- Fortunately, a bypass mechanism is available by using `immediate` attribute in command component.
- Default value of `immediate` attribute is False.

Example

- ```
  <h:commandButton value="Cancel"
  action="cancel" immediate="true"/>
  ```
  ```

Bean Validation

- JSF 2.0 integrates with the *Bean Validation Framework (JSR 303)*
- It is a general framework for specifying validation constraints.
- Validations are attached to fields or property getters of a Java class

Bean Validation Cont.

Example

```
• public class PaymentBean {
  @Size(min=13) private String card;
  @Future  private Date  myDate;
```

Bean Validation Cont.

- The advantage of using bean validation appears when you use the same bean in several pages.
- You don’t need to add validation rules to each page.
- Just add the validation to the bean and it will be applied whenever the bean used.

Bean Validation Messages

- You can override the default error messages.

Create a file ValidationMessages.properties in the default (root) package of your application.

Override the standard messages, for example:
javax.validation.constraints.Min.message =Must be at least {value}

Bean Validation Messages Cont.

- To provide a custom message for a particular validation:

Reference the bundle key in the message attribute:

```
@Size(min=13,
message="{com.corejsf.credit
CardLength}")
private String card = "";
```

Define the key in `ValidationMessages.properties`:

```
com.corejsf.creditCardLength=The
credit card number must have at
least 13 digits
```

Annotations in the Bean Validation Framework

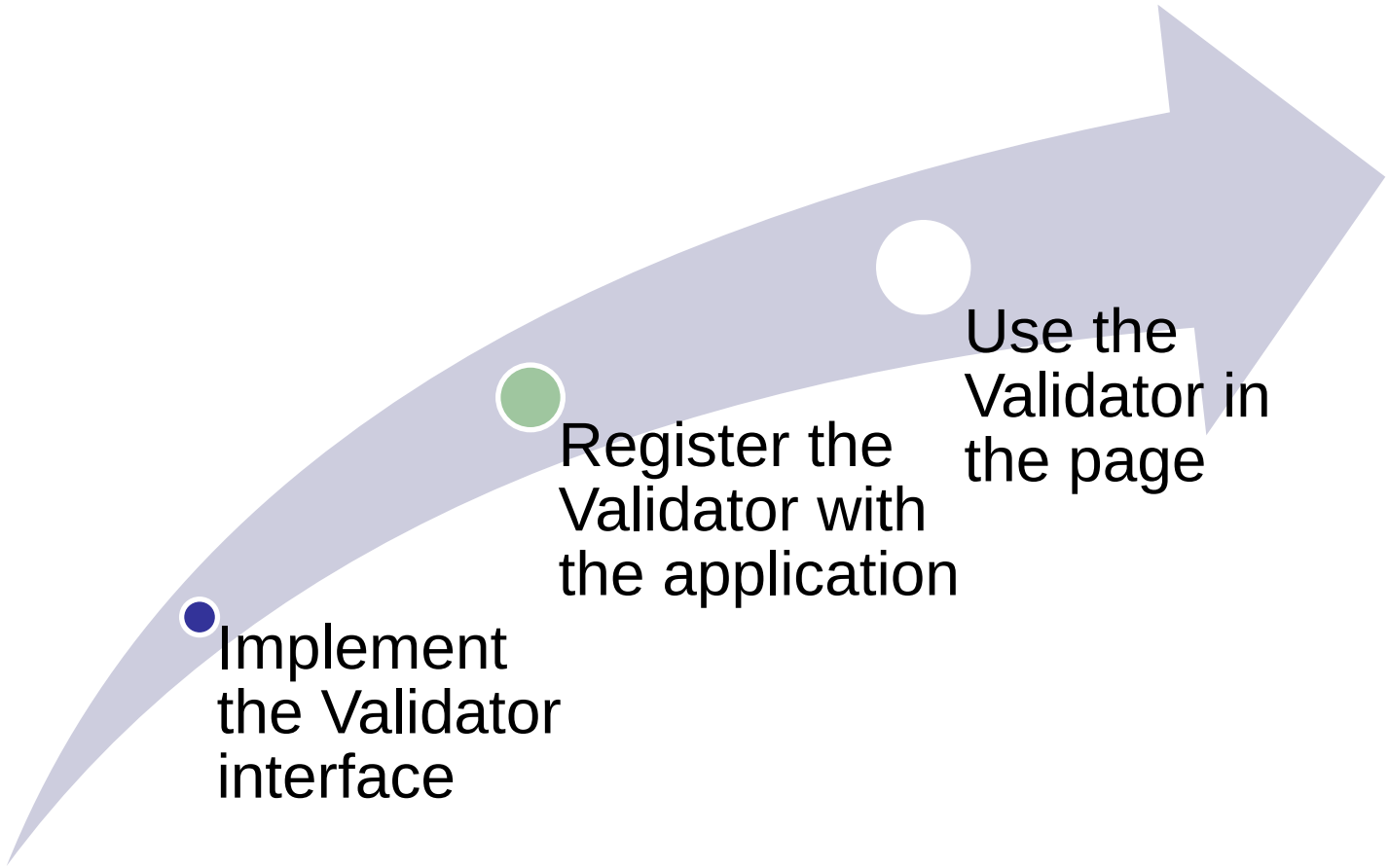
Annotation	Attribute	Purpose
@Null, @NotNull	None	Check that a value is null or not null.
@Min, @Max	The bound as a Long	Check that a value is at least or at most the given bound. The type must be one of int, long, short, byte and their wrappers, BigInteger, BigDecimal. Note: double and float are not supported due to roundoff.
@DecimalMin, @DecimalMax	The bound as a String	As above. Can also be applied to a String.
@Digits	integer, fraction	Check that a value has, at most, the given number of integer or fractional digits. Applies to int, long, short, byte and their wrappers, BigInteger, BigDecimal, String.
@Past, @Future	None	Check that a date is in the past or in the future.
@AssertTrue, @AssertFalse	None	Check that a Boolean value is true or false.

Annotations in the Bean Validation Framework Cont.

Annotation	Attribute	Purpose
@Size	min, max	Check that the size of a string, array, collection, or map is at least or at most the given bound.
@Pattern	regexp, flags	A regular expression and optional compilation flags.

Custom Validator

- **Steps for Creating/Using Custom Validator:**



Custom Validators Cont.

- **Step1: Implement Validator Interface**

```
void validate (FacesContext context,
  UIComponent component, Object  value)
      {
      if (validation fails)
            {
                  ……….
            }
      }
```

Custom Validators Cont.

- **Step2: Register validator**
  - You associate the ID with the validator in one of two ways.

@FacesValidator annotation

```
• @FacesValidator("com.corejsf.Card")
• public class CreditCardValidator implements
  Validator
```

Add entry to faces-config.xml

```
• <validator>
  <validator-id>com.corejsf.Card</validator-id>
  <validator-
  class>com.corejsf.CreditCardValidator</validator-
  class>
  </validator>
```

Custom Validators Cont.

- **Step3: Use the validator in the Page**
  - Now we can use the **f:validator** tag and specify the converter ID:

Example

```
<h:inputText id="card" value="#{payment.card}"
required="true">
<f:validator validatorId="com.corejsf.Card"/>
</h:inputText>
```

Reporting Custom Validator Errors

- When a validator detects an error, it should throw a ValidatorException.

Example

```
• if (validation fails) {
  FacesMessage message = ...;
  message.setSeverity(FacesMessage.SEVERITY_ERROR);
  throw new ValidatorException(message);
  }
```

Validating with Bean Methods

- Another way of custom validation is to add the validation method to an existing class and invoke it through a method expression.

Example

```
• <h:inputText id="card"
  value="#{payment.card}" required="true"
  validator="#{payment.check}" />
```

```
• public class PaymentBean {
  ...
  public void check(FacesContext context,
  UIComponent component, Object value) {
                                  ... // same
  code as in the preceding example
  }}
```

Display Error Messages

1. Add h:message tags whenever you use converters or validators.
2. Give an ID to the component to which the validator or converter attached.
3. Reference that ID in the h:message tag using for attribute.

Display Error Messages Cont.

Example

- ```
  <h:inputText id="amount"
  label="#{msgs.amount}"
  value="#{payment.amount}"/>
  <h:message for="amount"/>
  ```
  ```

Display All Error Messages

- **h:messages tag** used to show a listing of all messages from all components.

Example

- `<h:messages/>`

Changing the Text of Standard Error Messages

- Each validation or conversion error has a standard message that will be displayed if the validation or conversion failed.
- You can change the standard conversion and validation messages for your entire web application.

Changing the Text of Standard Error Messages

1-Set Up Message Bundle

- Creat a property file in the java sources directory of your application.

2-Add messages Definition to Bundle

- javax.faces.converter.NumberConverter.NUMBER_detail= "{0}" is not a number.

3-Register Bundle in faces-config.xml

- <application>
  <message-bundle> com.corejsf.messages</message-bundle>
  </application>

Lab Exercise

Assignments

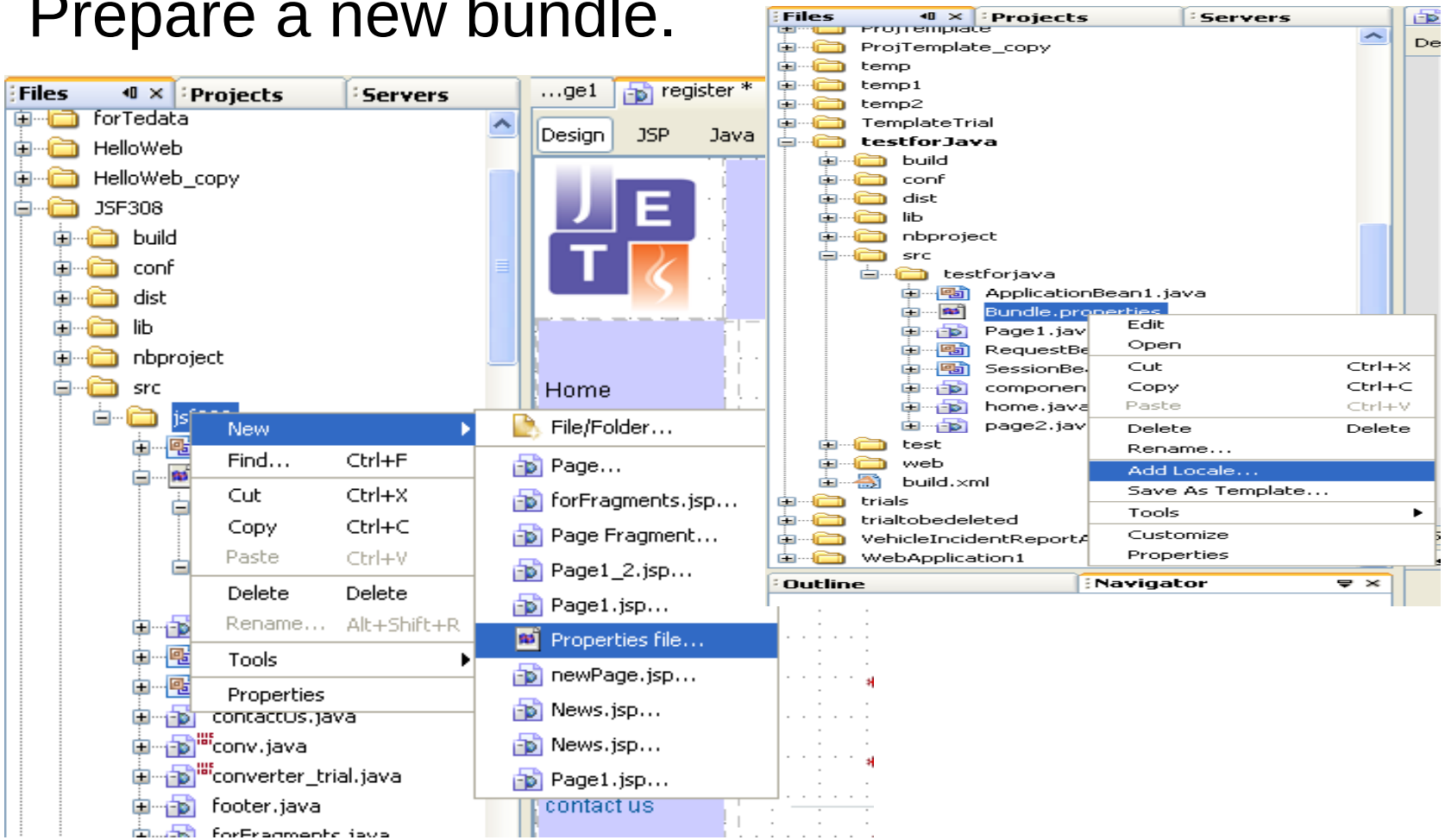
Registration Page

Input fields	Required	Validators	Converter
User name	All fields are required with custom message (apply internationalization (en_us, ar_eg)		
Password		f:validateLength	
Birthdate			f:convertDateTime
Email		Custom validator	
Credit_card_number			Custom converter

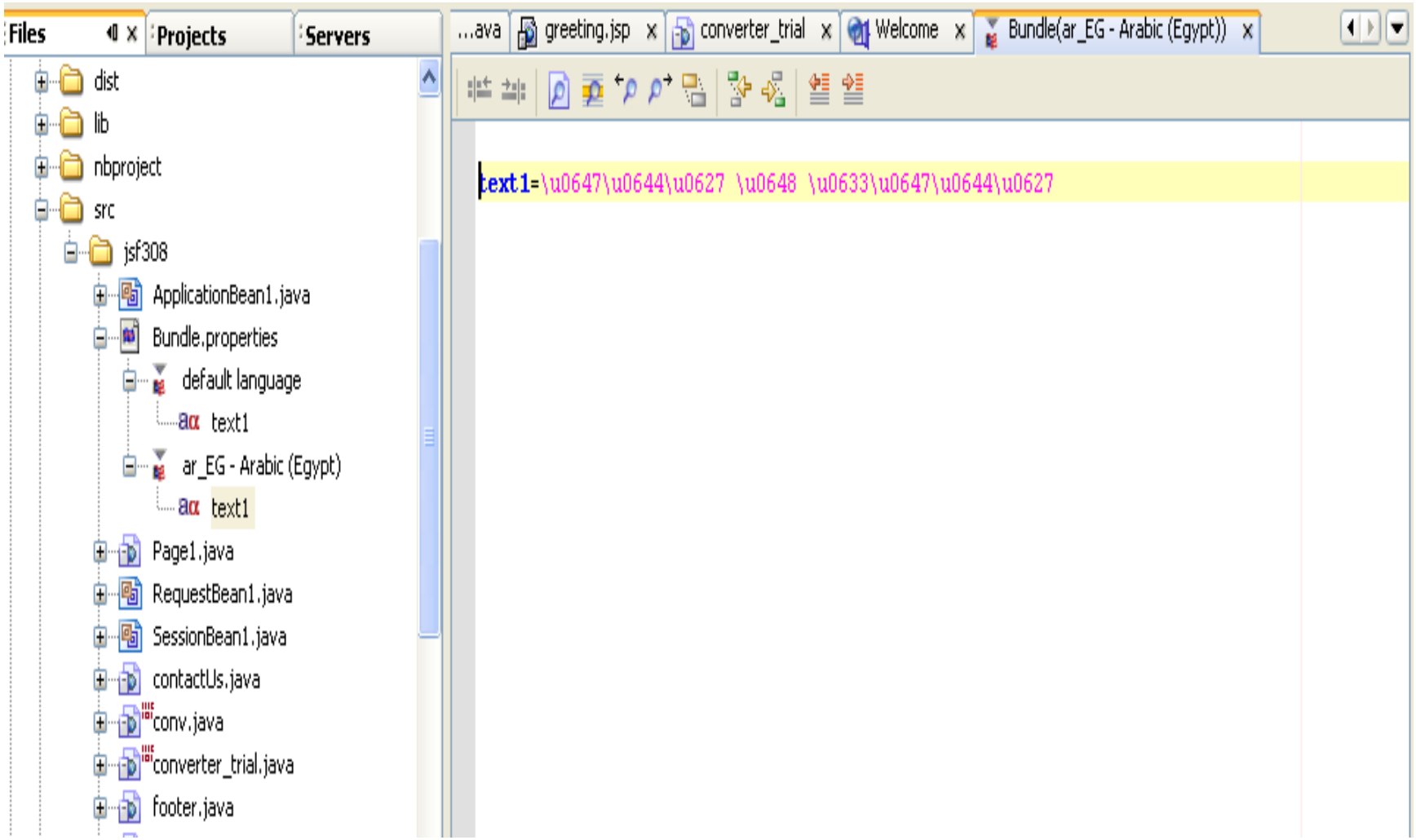
- **Submit to Display page.**
- **Cancel registration and back to welcome page**

How to apply internationalization?

- Prepare a new bundle.



How to apply internationalization? (cont'd)



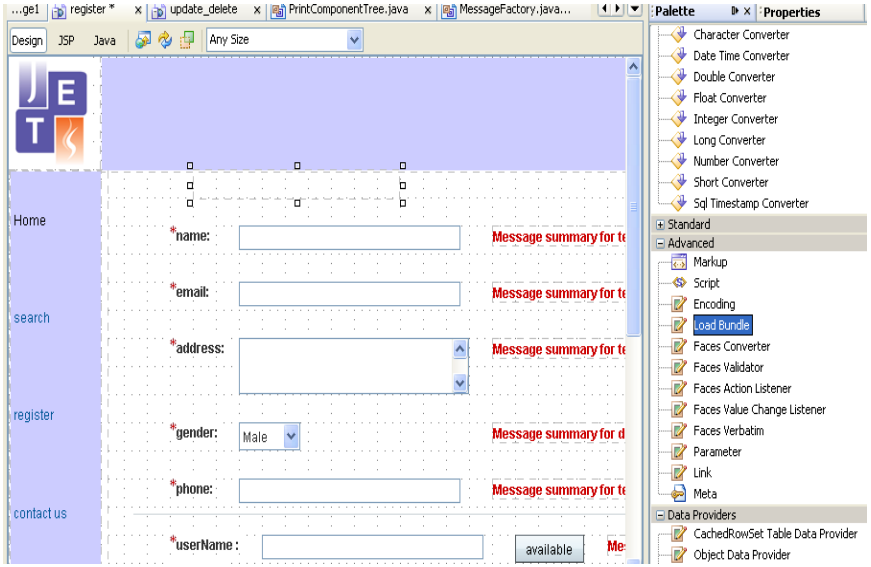
How to apply internationalization? (cont'd)

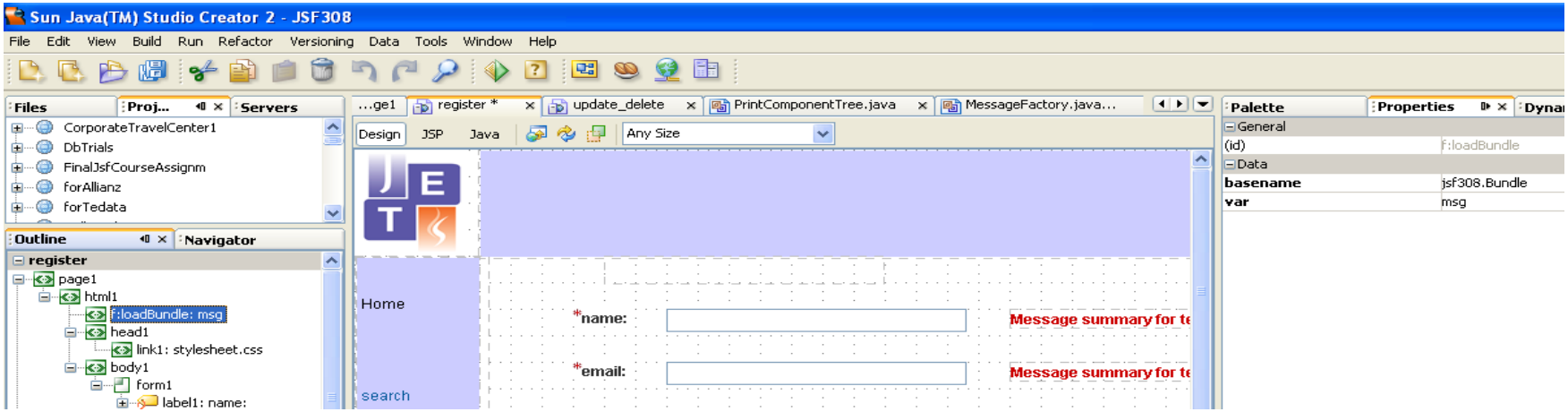
- Configure your bundle in faces-config.xml

```
<application>
    <locale-config>
        <default-locale>en</default-locale>
        <supported-locale>ar_EG</supported-locale>
    </locale-config>
    <resource-bundle>
        <base-name>jsf308.Bundle</base-name>
        <var>msg</var>
    </resource-bundle>
</application>
```

How to apply internationalization? (cont'd)

- Import the bundle in your page (add load- Bundle Component in your page) , and use it with your components





How to apply internationalization? (cont'd)

- **Localization tag:**

  <f:loadbundle> element:

  - Loads a resource bundle, stores properties as a Map
  - **Example:**

```
<f:loadBundle basename="myjsf.messages"
  var="msgs"/>
<h:outputText value="#{msgs.windowTitle}"/>
<h:outputText value="#{msgs.namePrompt}"/>
```

How to apply customized error messages?

- The key must be the full qualified name of the exception