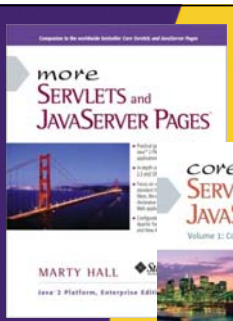
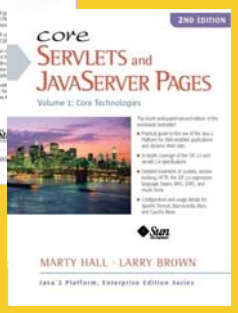
JSF 2: Handling GUI (Application) Events

JSF 2.2 Version

Originals of slides and source code for examples: http://www.coreservlets.com/JSF-Tutorial/jsf2/
Also see the PrimeFaces tutorial – http://www.coreservlets.com/JSF-Tutorial/primefaces/
and customized JSF2 and PrimeFaces training courses – http://courses.coreservlets.com/jsf-training.html





For live training on JSF 2, PrimeFaces, or other Java EE topics, email hall@coreservlets.com

Marty is also available for consulting and development support

Taught by the author of *Core Servlets and JSP*, this tutorial, and JSF 2.2 version of *Core JSF*. Available at public venues, or customized versions can be held on-site at your organization.

- Courses developed and taught by Marty Hall
  - JSF 2, PrimeFaces, Ajax, jQuery, Spring MVC, JSP, Android, general Java, Java 8 lambdas/streams, GWT, custom topic mix
  - Courses available in any location worldwide. Maryland/DC area companies can also choose afternoon/evening courses.
- Courses developed and taught by coreservlets.com experts (edited by Marty)
  - Hadoop, Spring, Hibernate/JPA, RESTful Web Services

Contact hall@coreservlets.com for details

Topics in This Section

- **Motivation**
- **Comparing action controllers to action listeners**
- **Action listeners**
- **Handling application events by using separate h:form**
- **Value change listeners**
- **Using JavaScript to submit form**
  - Dealing with browser incompatibilities

Overview

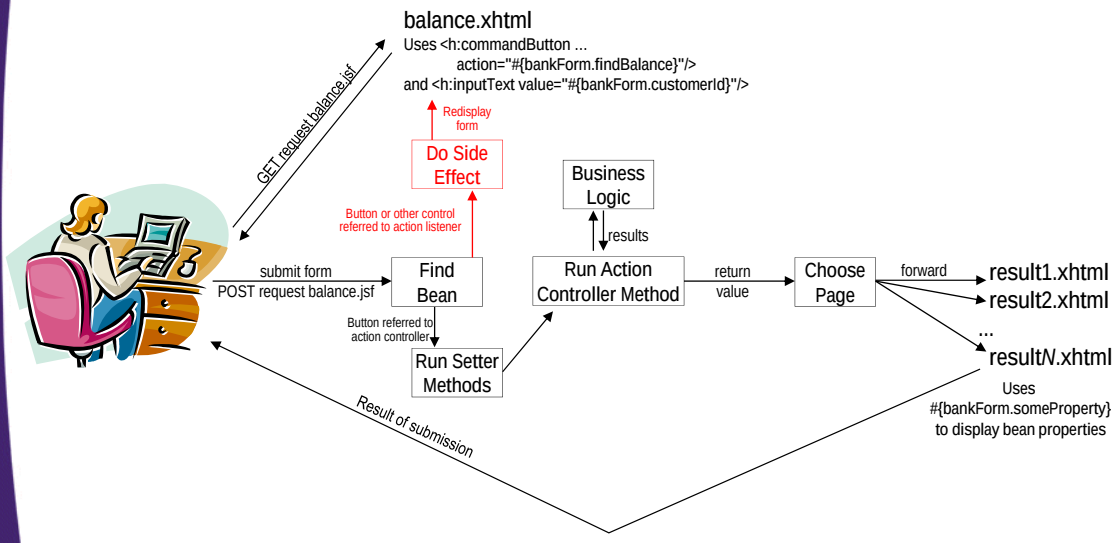
Motivation

- **There are 2 varieties of user interface events**
  - Events that start back-end processing
  - Events that affect only the format of the user interface
- **JSF categorizes code that handles these as *action controllers* and *event listeners***
  - Action controllers handle main form submission
    - Fire after bean has been populated (see previous section)
    - Fire after validation logic (see upcoming section)
    - Return strings that directly affect page navigation
  - Event listeners handle UI events
    - Often fire before bean has been populated
    - Often bypass validation logic
    - Never directly affect page navigation

Event Handlers vs. Ajax

- **Event handlers are one of the most convenient features of JSF**
  - Especially compared to Struts or regular MVC
- **However, Ajax is sometimes even better**
  - Some of the situations where event handlers apply result in only a small number of elements changing
  - In that case, Ajax often yields a more interactive user experience
    - If you need to update the entire page, use event listeners. If you need to update only part of the page, use Ajax.
  - See later lecture on Ajax support in JSF 2

JSF Flow of Control (Updated but Still Simplified)



Types of Event Listeners: ActionListener

- **Summary**
  - Fired by buttons, image maps, and hypertext links (links have attached JavaScript)
    - <h:commandButton value="..." .../>
    - <h:commandButton image="..." .../>
    - <h:commandLink .../>
  - These elements automatically submit the form
- **Syntax**
  - public void blah(ActionEvent e) { …}
    - With commandButton, the ActionEvent does not contain much useful info, so is usually ignored. But method signature requires that you declare it.
  - <h:commandButton … actionListener="#{bean.blah}" immediate="true"/>

Important Alternative: Putting Button in Separate Form

- **Button in same h:form as input elements**
  - <h:commandButton … actionListener="#{bean.blah}" immediate="true"/>
    - Using immediate prevents the setter methods from firing for the input elements, and blocks validation
    - The blah method must follow the method signature shown on previous page
- **Button in separate h:form**
  - <h:commandButton … action="#{bean.foo}"/>
    - This is in separate form, and that separate form has no input elements or validation anyhow
    - The foo method has normal action controller signature (zero args and returning String), and can just return null to tell JSF to redisplay original page after the method finishes.
    - Only downside is that button must be in location in page where it is possible to use separate form.

Types of Event Listeners: ValueChangeListener

- **Summary**
  - Fired by combo boxes, checkboxes, radio buttons, etc.
    - <h:selectOneMenu .../>
    - <h:selectBooleanCheckbox.../>
    - <h:selectOneRadio .../>
    - <h:inputText .../>
  - These elements do *not* automatically submit the form
- **Syntax**
  - public void blah(ValueChangeEvent e) { …}
    - The event has useful info in it: the value just selected
  - <h:selectOneRadio …onclick="submit()" valueChangeListener="#{bean.blah}"/>
- **Important alternative**
  - Often easier to omit the valueChangeListener and just have the "value" do something as a side effect, since these elements don't do page navigation anyhow

Action Listeners

Using ActionListener in Facelets

- **Some buttons submit the form and start backend processing**
  - Use <h:commandButton action="..." ...>
- **Other buttons affect only the UI**
  - Use <h:commandButton actionListener="..." .../>
  - You usually want this process to occur before beans are populated and especially before validation occurs
    - Since forms are often incomplete when the UI is adjusted
  - Use "immediate" to designate that listener fires before validation is performed or beans are populated

```
<h:commandButton actionListener="..."
                 immediate="true" .../>
```

  - All this is only necessary if button is in same h:form as the "real" form elements

Implementing ActionListener in the Java Code

- **Listener need not be in the form bean class**
  - Sometimes use a separate bean to represent GUI information. Often session scoped.
- **Method takes ActionEvent as argument**
  - void return type (*not* String as in action controllers)
  - ActionEvent is in javax.faces.event
  - ActionEvent has a getComponent method that lets you obtain the UIComponent reference
    - From the UIComponent, you can get the component ID, renderer, and other low-level information
  - Sample code

```
public void someMethod(ActionEvent event) {
  doSomeSideEffects();
}
```


Example

- **Idea**
  - Adapt the registration example from the I18N tutorial section. Provide pushbuttons to change the font size.
- **Approach**
  - Store the name of the body style to be used in the property of a session-scoped bean

```
<h:body styleClass="#{formSettings.bodyStyleClass}">
```

  - Provide two pushbuttons that run action listeners which change the body-style class

```
<h:commandButton value="#{msgs.normalFont}"
                 actionListener="#{formSettings.setNormalSize}"
                 immediate="true"/>
<h:commandButton value="#{msgs.largeFont}"
                 actionListener="#{formSettings.setLargeSize}"
                 immediate="true"/>
```

Input Form: Top (register1.xhtml)

```
<!DOCTYPE …>
<html …>
<h:head><title>#{msgs.registrationTitle}</title>
<link href="./css/styles.css"
      rel="stylesheet" type="text/css"/>
</h:head>
<h:body styleClass="#{formSettings.bodyStyleClass}">
…
<fieldset>
<legend>#{msgs.registrationTitle}</legend>
<h:outputFormat value="#{msgs.prompt}">
  <f:param value="#{msgs.firstName}"/>
</h:outputFormat>:
<h:inputText value="#{person.firstName}"/>
…
<h:commandButton value="#{msgs.buttonLabel}"
                 action="#{person.doRegistration}"/>
</fieldset>
```

There is a second way to load CSS files in JSF 2. See section on relocatable resources in Page Templating tutorial. However, the simple/basic usage is fine here.

This is a "normal" pushbutton as shown many times in earlier sections. It results in the normal request processing flow.

Input Form: Bottom (register1.xhtml)

```
<br/>
<div align="center">
<h:commandButton value="#{msgs.normalFont}"
                 actionListener="#{formSettings.setNormalSize}"
                 immediate="true"/>
<h:commandButton value="#{msgs.largeFont}"
                 actionListener="#{formSettings.setLargeSize}"
                 immediate="true"/>
</div>
</h:form>
</h:body></html>
```

These two pushbuttons invoke action listeners. So, pressing them results in going to the server, calling setNormalSize or setLargeSize, then redisplaying the form.

Bean for Action Listener (Bean for Form Settings)

```
@ManagedBean
@SessionScoped
public class FormSettings implements Serializable {
  private boolean isNormalSize = true;

  public String getBodyStyleClass() {
    if (isNormalSize) {
      return("normalSize");
    } else {
      return("largeSize");
    }
  }
  public void setNormalSize(ActionEvent event) {
    isNormalSize = true;
  }
  public void setLargeSize(ActionEvent event) {
    isNormalSize = false;
  }
…}
```


Bean for Action Controller ("Main" Bean)

```
@ManagedBean
public class Person {
  private String firstName, lastName, emailAddress;

  // Accessor methods: getFirstName, setFirstName, etc.

  public String doRegistration() {
    if (isAnyEmpty(firstName, lastName, emailAddress)) {
      return("missing-input");
    } else {
      return("confirm-registration");
    }
  }

  …
}
```

This action controller method is called by the "normal" button in the top half of the form, resulting in the normal request processing flow.

faces-config.xml

```
<?xml version="1.0"?>
<faces-config …
   version="2.0">
  <application>
    <resource-bundle>
      <base-name>messages</base-name>
      <var>msgs</var>
    </resource-bundle>
  </application>
</faces-config>
```

The first example uses strings from properties files, but not I18N. So, the only file used is messages.properties. The second example will also set the Locale and make use of messages_es.properties.

Default Properties File (messages.properties)

```
registrationTitle=Registration
firstName=First Name
lastName=Last Name
emailAddress=Email Address
registrationText=Please Enter Your {0}, {1}, and {2}.
prompt=Enter {0}
buttonLabel=Register Me
successTitle=Success
successText=You Registered Successfully.
switchLanguage=En Español
normalFont=Normal Font
largeFont=Large Font
errorTitle=Error!
missingData=Missing input. Please try again.
```

In Eclipse, this is src/messages.properties. In the deployed project, it is WEB-INF/classes/messages.properties. The basic use of properties files was discussed in an earlier tutorial section.

CSS File

```
.normalSize { font-size: 110% }
.largeSize { font-size: 200% }
…
```

These are the two names returned by getBodyStyleClass in the FormSettings bean.

Results Page: Success (confirm-registration.xhtml)

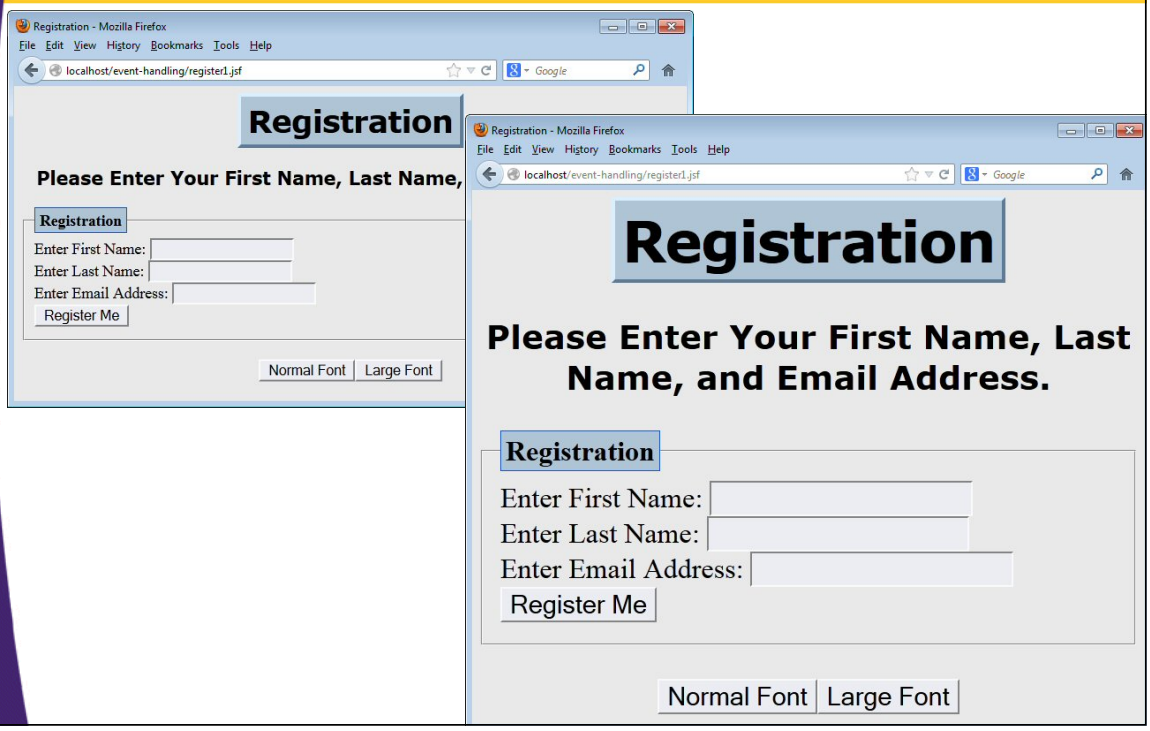
```
<!DOCTYPE …>
<html …>
<f:view locale="#{formSettings.locale}">
<h:body styleClass="#{formSettings.bodyStyleClass}">
<h1 class="title">#{msgs.successTitle}</h1>
<h3>#{msgs.successText}</h3>
<ul>
  <li>#{msgs.firstName}: #{person.firstName}</li>
  <li>#{msgs.lastName}: #{person.lastName}</li>
  <li>#{msgs.emailAddress}: #{person.emailAddress}</li>
</ul>
</h:body></f:view></html>
```

This example does not make use of the f:view locale settings, but the results pages are shared with the next two examples, which do make use of this. This setting will be explained in next example.

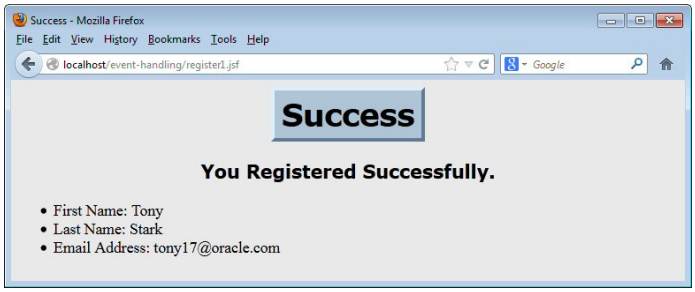
Results Page: Error (missing-input.xhtml)

```
<!DOCTYPE …>
<html …>
<f:view locale="#{formSettings.locale}">
…
<h:body styleClass="#{formSettings.bodyStyleClass}">
<h1 class="title">#{msgs.errorTitle}</h1>
<h1>#{msgs.missingData}</h1>
…
</h:body></f:view></html>
```


Results: Input Form



Results: Results Pages (Success)



Results: Results Pages (Error)

Changing the Locale Programmatically

Setting the Locale from User Settings

- **Using browser settings (last section)**
  <f:view locale="#{facesContext.externalContext.requestLocale}">
  - Issues
    - Many users do not set the language in their browser. Major problem, since we cannot detect this.
    - This can send unsupported language choices. Minor problem: we just make blah.properties (with no language name) have default language choices.
- **Based on user choices (now)**
  <f:view locale="#{userPreferences.currentLocale}">
  - Issues
    - Need way to let user choose. That is the main topic here: redisplaying form after the user chooses.
    - Should persist from page to page. Use session scope.

Example

- **Idea**
  - Adapt the previous example. Provide pushbutton to switch between English and Spanish.
- **Approach**
  - Create two properties files
    - messages.properties and messages_es.properties
  - Provide pushbutton that runs action listener which changes the Locale
    - Sets a flag that will determine what getLocale returns
  - Look up the Locale in f:view, as in I18N tutorial
    - <f:view locale="#{formSettings.locale}">

Input Form (register2.xhtml)

```
<!DOCTYPE …>
<html …>
<f:view locale="#{formSettings.locale}">
…
<!-- Same as register1.xhtml except for f:view
     and the button below. -->
<h:commandButton value="#{msgs.switchLanguage}"
        actionListener="#{formSettings.swapLocale1}"
        immediate="true"/>
…
</f:view></html>
```

Bean for Action Listener (Part Called by f:view)

```
@ManagedBean
@SessionScoped
public class FormSettings implements Serializable {
  … // Code for font size control shown earlier
  private boolean isEnglish = true;
  private static final Locale ENGLISH = new Locale("en");
  private static final Locale SPANISH = new Locale("es");
  private Locale locale = new Locale("en");

  public Locale getLocale() {
    return(locale);
  }
…
```


Bean for Action Listener (Part Called by ActionListener)

```
  public void swapLocale(ActionEvent event) {
    isEnglish = !isEnglish;
    if (isEnglish) {
      locale = ENGLISH;
    } else {
      locale = SPANISH;
    }
  }
```

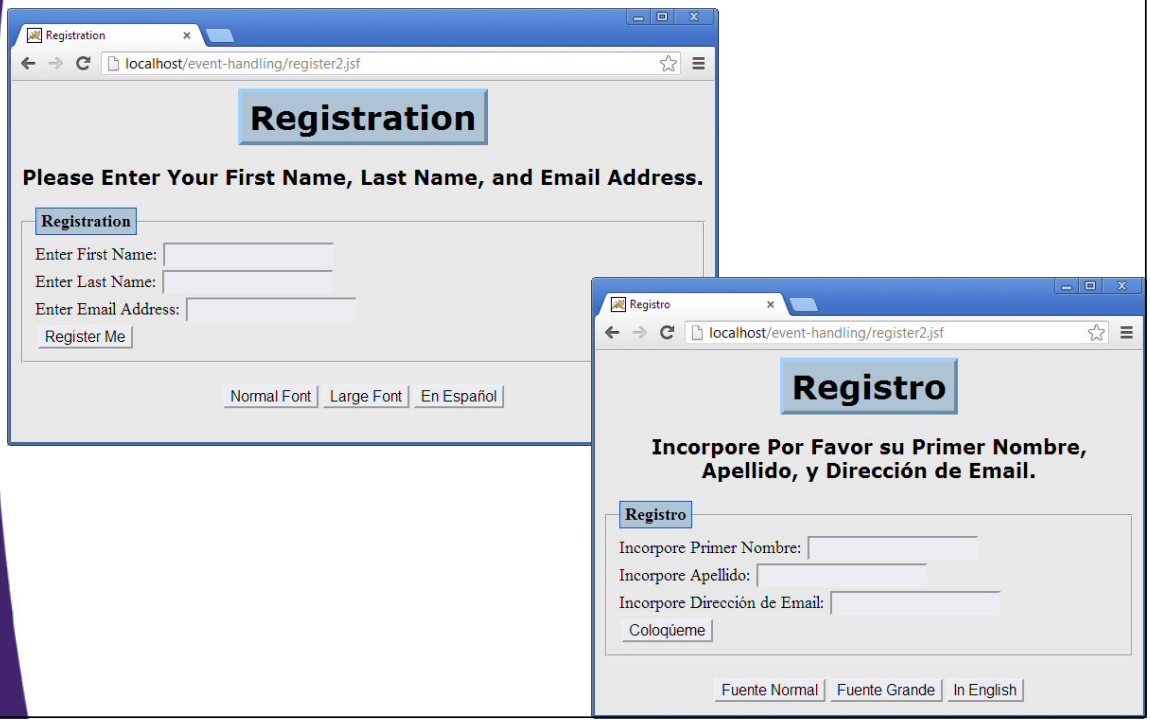
Previously-Shown Files

- **Unchanged from last example**
  - Main bean
    - Person, with the doRegistration action controller method
  - faces-config.xml
    - Defines the msgs variable as referring to messages.properties
  - messages.properties
    - Default/English strings
  - CSS file
    - Defines normalSize and largeSize class names
  - Results pages
    - confirm-registration.xhtml and missing-input.xhtml
    - These also use <f:view locale="#{formSettings.locale}"/>

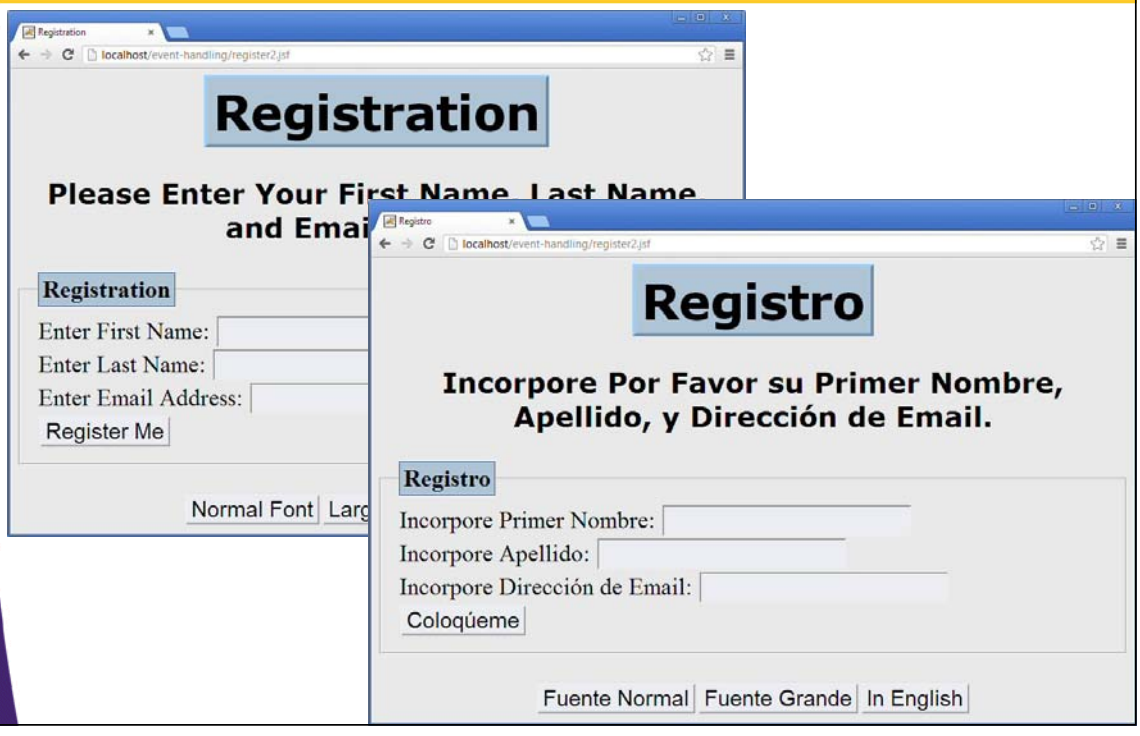
Spanish Properties File (messages_es.properties)

```
registrationTitle=Registro
firstName=Primer Nombre
lastName=Apellido
emailAddress=Dirección de Email
registrationText=Incorpore Por Favor su {0}, {1}, y {2}.
prompt=Incorpore {0}
buttonLabel=Coloqúeme
successTitle=Éxito
successText=Se Registró con Éxito.
switchLanguage=In English
normalFont=Fuente Normal
largeFont=Fuente Grande
errorTitle=¡Error!
missingData=Falta de input. Por favor, inténtelo de nuevo.
```

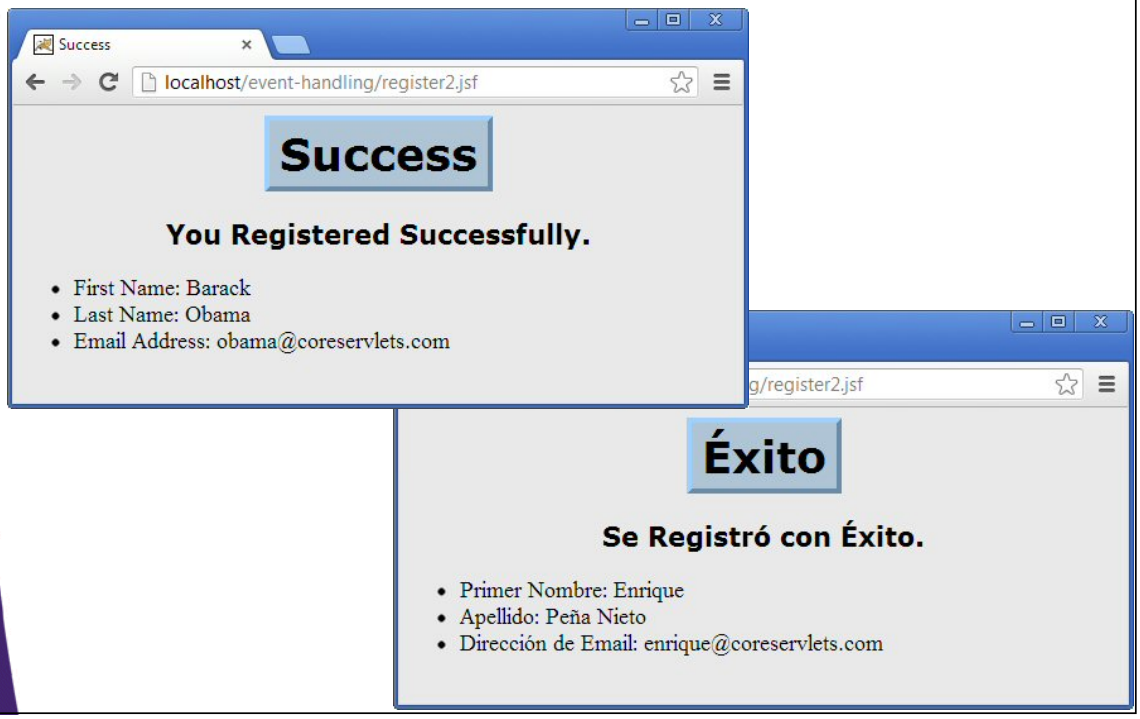
Results: Input Form (Normal Font Size)



Results: Input Form (Large Font Size)



Results: Results Pages (Success)



Results: Results Pages (Error)

Using Input Elements that Don't Normally Submit Forms

Option 1: Use ValueChangeListener

- **Before: ActionListener was used**
  - For buttons. Form was automatically submitted when clicked
- **Now: ValueChangeListener is used**
  - For combobox, listbox, radio button, checkbox, textfield, etc.
- **Difference from ActionListener**
  - Form not automatically submitted
  - Need to add JavaScript to submit the form
    `onclick="submit()"` or `onchange="submit()"`
  - Event incompatibility between Firefox and IE
    - Firefox, Netscape, and Opera fire the onchange events when the combobox selection changes, radio button is selected, or checkbox is checked/unchecked
    - Internet Explorer fires event when selection changes *and* another GUI control receives the input focus
      - onclick generally works consistently in current browsers. Older IE versions behave differently. Test on multiple browsers!

Implementing ValueChangeListener in Java Code

- **Listener can be in session-scoped bean**
  - Different from "main" bean, as discussed previously
- **Method takes ValueChangeEvent as arg**
  - Useful ValueChangeEvent methods
    - getComponent (as mentioned for ActionEvent)
    - getOldValue (previous value of GUI element)
    - getNewValue (current value of GUI element)
      - Needed since bean has probably not been populated
      - Value for checkbox is of type Boolean
      - Value for radio button or textfield corresponds to request parameter
  - Sample code

```
public void someMethod(ValueChangeEvent event) {
  Boolean flag = (Boolean)event.getNewValue();
  takeActionBasedOn(flag);
}
```


Option 2: Have "value" Perform Side Effect

- **Facelets code**

```
<h:selectOneMenu value="#{bean.someProperty}"
                 onchange="submit()"
                 immediate="true">
  <f:selectItems value="#{bean.choices}"/>
</h:selectOneMenu>
```

- **Java code**

```
public void setSomeProperty(String prop) {
  this.prop = prop;
  doSomeOtherSideEffects();
}
public String getSomeProperty() {
  return(prop);
}
```

Example

- **Idea**
  - Extend previous example. Use radio buttons to switch among English, Spanish, and Japanese
- **Approach**
  - Have radio button value change Locale as side effect
    - Facelets

      <h:selectOneRadio value="#{formSettings.language}" …>
    - Java

      public void setLanguage(String language) {
        this.language = language;
        locale = new Locale(language);
      }
  - Make sure clicking radio button submits the form
    - onclick="submit()"

Input Form (register3.xhtml)

```
<!DOCTYPE …>
<html …>
<f:view locale="#{formSettings.locale}">
…
<!-- Same as register2.xhtml except
     h:commandButton replaced by h:selectOneRadio -->
<h:selectOneRadio value="#{formSettings.language}"
                  onclick="submit()"
                  immediate="true">
  <f:selectItems value="#{formSettings.languages}"/>
</h:selectOneRadio>
…
</f:view></html>
```

Bean: Processing the “value”

```
private String language = "en";

public String getLanguage() {
  return(language);
}

public void setLanguage(String language) {
  this.language = language;
  locale = new Locale(language);
}
```

What is displayed to the user is the language name (“English”, “Español”, etc.). But what is returned as the value is the locale name (“en”, “es”, etc.). As discussed earlier, this can be accomplished with menus and radio buttons by using a Map as the value. The Map keys are the display values and the Map values are the return values. See next page for the code.

Bean: The List of Choices

```
private static final Map<String,String> LANGUAGE_MAP =
  new LinkedHashMap<String,String>();

static {
  LANGUAGE_MAP.put("English", "en");
  LANGUAGE_MAP.put("Español", "es");
  LANGUAGE_MAP.put("日本人", "jp");
}

public Map<String,String> getLanguages() {
  return(LANGUAGE_MAP);
}
```

getLanguages is what is called by f:selectItems in the facelets code

messages_jp.properties

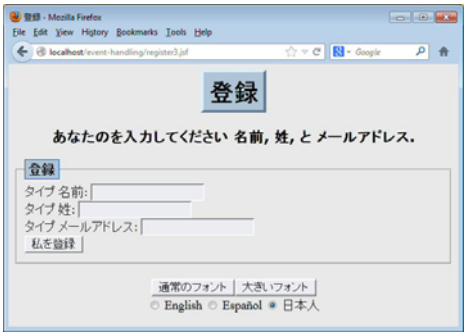
```
registrationTitle=\u767B\u9332
firstName=\u540D\u524D
lastName=\u59D3
emailAddress=\u30E1\u30FC\u30EB\u30A2\u30C9\u30EC\u30B9
registrationText=\u3042\u306A\u305F\u306E\u3092\u5165\u529B\u3057\u3066\u304F\u3060\u3055\u3044 {0}, {1}, \u3068 {2}.
prompt=\u30BF\u30A4\u30D7 {0}
buttonLabel=\u79C1\u3092\u767B\u9332
successTitle=\u6210\u529F
...
```

Eclipse automatically converts to Unicode escaped characters when you paste in non-ISO-Latin-1 characters (in this case, from Google Translate).

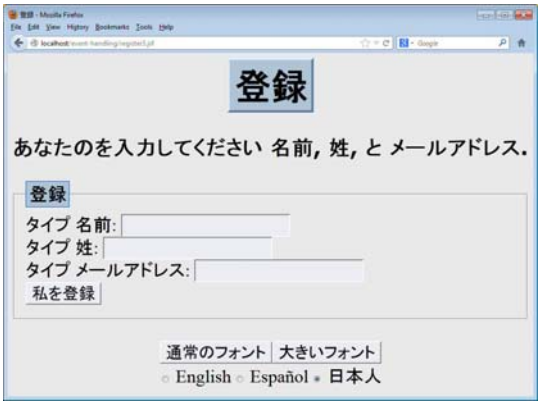
Previously-Shown Files

- **Unchanged from last example**
  - Main bean
    - Person, with the doRegistration action controller method
  - faces-config.xml
    - Defines the msgs variable as referring to messages.properties
  - English and Spanish properties files
    - messages.properties and messages_es.properties
  - CSS file
    - Defines normalSize and largeSize class names
  - Results pages
    - confirm-registration.xhtml and missing-input.xhtml

Results: Input Form (Normal Font Size)



Results: Input Form (Large Font Size)



Setting Locale in JSF 2.0 (Fixed in 2.1 & 2.2)

Setting the Locale from User Settings (Continued)

- **Problem in JSF 2.0 [many Java EE servers]**
  - Calling setLocale on the view
    - Triggered when you reload the page
    - Not triggered when you redisplay the page after running action listener (or submitting form)
  - Setting the Locale of the UIViewRoot
    - Triggered when you redisplay the page after running action listener (or submitting form)
    - Not triggered when you reload the page
      - Because the Locale is reset to the default
- **Solution**
  - Do *both*!
    - Before navigating to page, call FacesContext.getCurrentInstance(). getViewRoot().setLocale(currentLocale);
    - On page, use <f:view locale="#{formSettings.currentLocale}">

Fixed in JSF 2.1. Works properly now.

Bean for Action Listener (Part Called by ActionListener)

```
public void swapLocale(ActionEvent event) {
   isEnglish = !isEnglish;
   if (isEnglish) {
     locale = ENGLISH;
   } else {
     locale = SPANISH;
   }
   FacesContext.getCurrentInstance().getViewRoot()
               .setLocale(locale);
 }
```

Red part NOT needed as of JSF 2.1.

Wrap-Up

Summary

- **Event listeners are used to handle events that affect only the user interface**
  - Should fire before beans are populated and validation is performed
    - Use immediate="true" to designate this behavior
    - Or, just put button in separate h:form element and return null
  - Form is redisplayed after listeners fire
    - No navigation rules apply
- **Things that submit forms:**
  - Use actionListener
    - Applies to buttons, hypertext links, or image maps
- **Things that don't submit forms**
  - Use side effect of "value" or use valueChangeListener
    - Applies to radio buttons, menus, list boxes, checkboxes, textfields, etc.
  - Need onclick="submit()" or onchange="submit()" to submit form
    - Test carefully on all expected browsers

Questions?

More info:
http://www.coreservlets.com/JSF-Tutorial/jsf2/ – JSF 2.2 tutorial
http://www.coreservlets.com/JSF-Tutorial/primefaces/ – PrimeFaces tutorial
http://courses.coreservlets.com/jsf-training.html – Customized JSF and PrimeFaces training courses
http://coreservlets.com/ – JSF 2, PrimeFaces, Java 7 or 8, Ajax, jQuery, Hadoop, RESTful Web Services, Android, HTML5, Spring, Hibernate, Servlets, JSP, GWT, and other Java EE training