Managed Beans I – Classes to Represent Form Info

Originals of Slides and Source Code for Examples:
http://www.coreservlets.com/JSF-Tutorial/jsf2/

Topics in This Section

- **Basic beans and "managed" beans**
- **Three parts of beans in JSF**
  - Getter/setter methods to represent input elements
  - Action controller method
  - Placeholder for results (properties derived from input)
- **Prepopulating input fields**
  - And building comboboxes (drop down menus)

Basic Beans and "Managed" Beans

Background: Basic Beans

- **Java classes that follow certain conventions**
  - Must have a zero-argument (empty) constructor
    - You can satisfy this requirement either by explicitly defining such a constructor or by omitting all constructors
  - Should have no public instance variables (fields)
    - You should already follow this practice and use accessor methods instead of allowing direct access to fields
  - Persistent values should be accessed through methods called get*Blah* and set*Blah*
    - If class has method getTitle that returns a String, class is said to have a String *property* named title
      - JSF uses #{book.title} to mean "call getTitle on 'book' ".
    - Boolean properties may use is*Blah* instead of get*Blah*
    - What matters is method name, not instance variable name

More on Bean Properties

- **Usual rule to turn method into property**
  - Drop the word "get" or "set" and change the next letter to lowercase. Again, instance var name is irrelevant.
    - Method name: getFirstName
    - Property name: firstName
    - Example: #{customer.firstName}
- **Exception 1: boolean properties**
  - If getter returns boolean or Boolean
    - Method name: getPrime or isPrime
    - Property name: prime
    - Example: #{myNumber.prime}
- **Exception 2: consecutive uppercase letters**
  - If two uppercase letters in a row after "get" or "set"
    - Method name: getURL
    - Property name: URL (not uRL)
    - Example: #{webSite.URL}

If you write the methods, it is considered better practice to avoid the consecutive uppercase letters, and to call the method getUrl, not getURL.

Bean Properties: Examples

Method Names	Property Name	Example JSF Usage
getFirstName setFirstName	firstName	#{customer.firstName} <h:inputText value="#{customer.firstName}"/>

isExecutive
setExecutive
(boolean property)
	executive	#{customer.executive}
		<h:selectBooleanCheckbox
		value="#{customer.executive}"/>

getExecutive
setExecutive
(boolean property)
	executive	#{customer.executive}
		<h:selectBooleanCheckbox
		value="#{customer.executive}"/>

getZIP setZIP	ZIP	#{address.ZIP} <h:inputText value="#{address.ZIP}"/>

Note 1: property name does not exist anywhere in your code. It is just a shortcut for the method name. Instance variable name is irrelevant.
Note 2: if you can choose the method names, it is a better practice to avoid consecutive uppercase letters.
E.g., use getZip and getUrl, not getZIP and getURL.

Why You Should Use Accessors, Not Public Fields

- **Bean rules**
  - To be a bean, you should use accessors, not public fields
    - Wrong

```
public double speed;
```

    - Right

```
private double speed;  // Var name need not match method name

public double getSpeed() {
    return(speed);
}

public void setSpeed(double speed) {
    this.speed = speed;
}
```

Note: in Eclipse, after you create instance variable, if you R-click and choose "Source", it gives you option to generate getters and setters for you.

- **OOP design**
  - You should do this in *all* your Java code anyhow. Why?

Why You Should Use Accessors, Not Public Fields

- 1) You can put constraints on values

```
public void setSpeed(double newSpeed) {
  if (newSpeed < 0) {
     sendErrorMessage(...);
     newSpeed = Math.abs(newSpeed);
   }
   speed = newSpeed;
}
```

  – If users of your class accessed the fields directly, then they would each be responsible for checking constraints.

Why You Should Use Accessors, Not Public Fields

- 2) You can change your internal representation without changing interface

```
// Instance var changed to store
// metric units (kph, not mph)

public void setSpeed(double newSpeed) {   // MPH
   speedInKph = convertMphToKph(newSpeed);
}

public void setSpeedInKph(double newSpeed) {
   speedInKph = newSpeed;
}
```

Why You Should Use Accessors, Not Public Fields

- **3) You can perform arbitrary side effects**

```
public double setSpeed(double newSpeed) {
  speed = newSpeed;
  updateSpeedometerDisplay();
}
```

  – If users of your class accessed the fields directly, then they would each be responsible for executing side effects. Too much work and runs huge risk of having display inconsistent from actual values.

Basic Beans: Bottom Line

- **It is no onerous requirement to be a “bean”**
  – You are probably following most of the conventions already anyhow
    - Zero arg constructor
    - No public instance variables
    - Use getBlah/setBlah or isBlah/setBlah naming conventions
- **JSF often refers to “bean properties”**
  – Which are shortcuts for getter/setter methods
    - getFirstName method: refer to “firstName”
      – #{customer.firstName}
    - isVeryCool method (boolean): refer to “veryCool”
      – #{invention.veryCool}
    - getHTMLString method: refer to “HTMLString”
      – #{message.HTMLString}

Managed Beans

- **JSF automatically "manages" the bean**
  - Instantiates it
    - Thus the need for a zero-arg constructor
  - Controls its lifecycle
    - Scope (request, session, application) determines lifetime
  - Calls setter methods
    - I.e., for <h:inputText value="#{customer.firstName"/>, when form submitted, the value is passed to setFirstName
  - Calls getter methods
    - #{customer.firstName} results in calling getFirstName
- **Declaring beans**
  - Simplest: @ManagedBean before class
    - Results in request scope. See next lecture for other scopes.
  - Most powerful: <managed-bean> in faces-config.xml
    - See separate section on navigation and faces-config.xml

Performance Principle: Make Getter Methods Fast

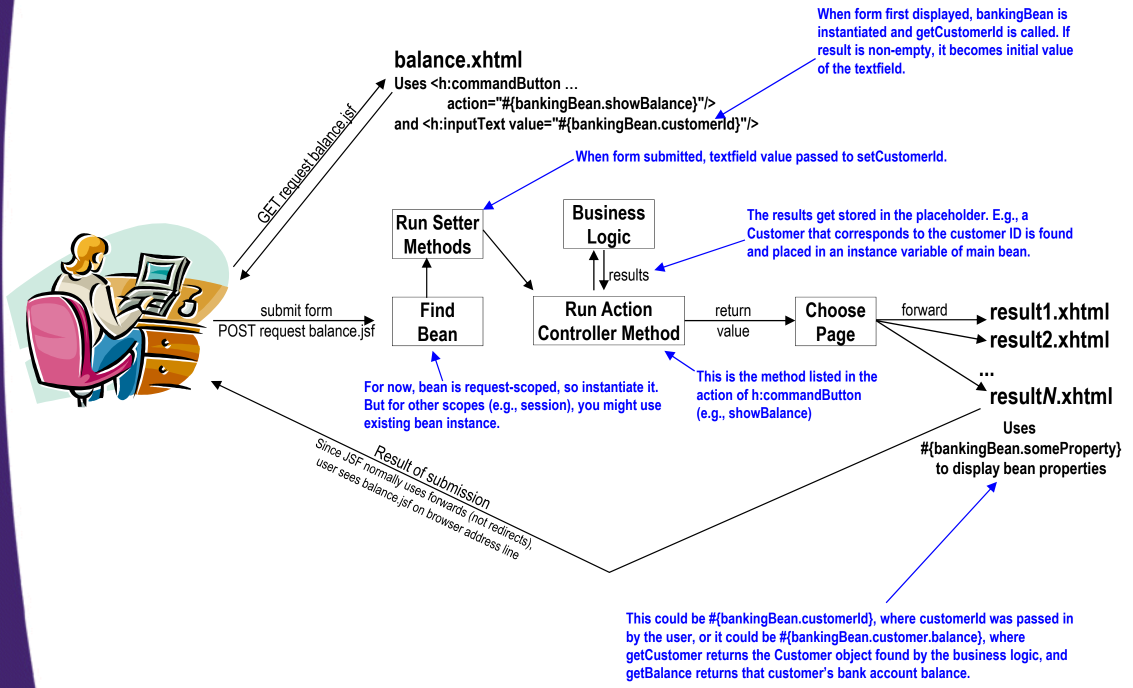
- **Problem**
  - Getter methods of managed beans called several times.
    - E.g., at a minimum, once when form is displayed (<h:inputText value="#{user.customerId}"/>) and again when result is shown (#{user.customerId}).
      - But often extra times.
  - So, if getter method talks to database or does other expensive operation, performance can be unexpectedly bad.
- **Solution**
  - Store data in instance variables, have getter methods merely return the existing values.

The Three Parts of Managed Beans

Overview

- **Managed beans typically have three parts**
  - Bean properties (i.e, pairs of getter and setter methods)
    - One pair for each input element
    - Setter methods called automatically by JSF when form submitted. Called *before* action controller method.
  - Action controller methods
    - Often only one, but could be several if the same form has multiple buttons
    - Action controller method (corresponding to the button that was pressed) called automatically by JSF
  - Placeholders for results data
    - Not automatically called by JSF: to be filled in by action controller method based on results of business logic.
    - Needs a getter method so value can be output in results page, but not required to have a setter method.

JSF Flow of Control (Updated but Still Simplified)


When form first displayed, bankingBean is instantiated and getCustomerId is called. If result is non-empty, it becomes initial value of the textfield.
balance.xhtml
Uses <h:commandButton …
action="#{bankingBean.showBalance}"/>
and <h:inputText value="#{bankingBean.customerId}"/>
GET request balance.jsf
When form submitted, textfield value passed to setCustomerId.
Run Setter Methods
Business Logic
results
The results get stored in the placeholder. E.g., a Customer that corresponds to the customer ID is found and placed in an instance variable of main bean.
submit form
POST request balance.jsf
Find Bean
Run Action Controller Method
return value
Choose Page
forward
result1.xhtml
result2.xhtml
...
resultN.xhtml
For now, bean is request-scoped, so instantiate it. But for other scopes (e.g., session), you might use existing bean instance.
This is the method listed in the action of h:commandButton (e.g., showBalance)
Uses #{bankingBean.someProperty} to display bean properties
Result of submission
Since JSF normally uses forwards (not redirects), user sees balance.jsf on browser address line
This could be #{bankingBean.customerId}, where customerId was passed in by the user, or it could be #{bankingBean.customer.balance}, where getCustomer returns the Customer object found by the business logic, and getBalance returns that customer's bank account balance.

Example

- **Idea**
  - Enter a bank customer id and a password
  - Get either
    - Page showing first name, last name, and balance
      - Three versions depending on balance
    - Error message about missing or invalid data
- **What managed bean needs**
  - Bean properties corresponding to input elements
    - I.e., getCustomerId/setCustomerId, getPassword/setPassword
  - Action controller method
    - Maps a customer ID to a Customer and stores the Customer in an instance variable
  - Placeholder for results data
    - An initially empty instance variable (for storing the Customer) and associated getter method.

Input Form (bank-lookup.xhtml)

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html">
…
<h:body>
…
<fieldset>
<legend>Bank Customer Lookup (Request Scope)</legend>
<h:form>
  Customer ID:
  <h:inputText value="#{bankingBean.customerId}"/><br/>
  Password:
  <h:inputSecret value="#{bankingBean.password}"/><br/>
  <h:commandButton value="Show Current Balance"
                   action="#{bankingBean.showBalance}"/>
</h:form>
</fieldset>
…
</h:body></html>
```

This value plays a dual role. When form is first displayed, bankingBean is instantiated and getCustomerId is called. If the value is non-empty, that result is the initial value of the textfield. Otherwise, the textfield is initially empty. When the form is submitted, bankingBean is reinstantiated (since it is request scoped, not session scoped) and the value in the textfield is passed to setCustomerId.

BankingBean.java: Part 1 (Bean Properties for Input Elements)

```
@ManagedBean
public class BankingBean  {
  private String customerId, password;

  public String getCustomerId() {
    return(customerId);
  }
  public void setCustomerId(String customerId) {
    this.customerId = customerId.trim();
    if (this.customerId.isEmpty()) {
      this.customerId = "(none entered)";
    }
  }
  public String getPassword() {
    return(password);
  }
  public void setPassword(String password) {
    this.password = password;    }
```

Called by JSF when form first displayed. Since it returns null in that case, the textfield is left blank initially.

When form submitted, the bean is instantiated again (since it is request-scoped, not session-scoped) and the value in the textfield is passed to this method.

getPassword and setPassword mostly have the same behavior as getCustomerId and setCustomerId above, but with the exception that the return value of getPassword does not affect the initial value of the password field, since browsers do not let you prefill values in password fields.

BankingBean.java: Part 2 (Action Controller Method)

```
private static CustomerLookupService lookupService =
  new CustomerSimpleMap();

public String showBalance() {
  if (!password.equals("secret")) {
    return("wrong-password");
  }
  customer = lookupService.findCustomer(customerId);
  if (customer == null) {
    return("unknown-customer");
  } else if (customer.getBalance() < 0) {
    return("negative-balance");
  } else if (customer.getBalance() < 10000) {
    return("normal-balance");
  } else {
    return("high-balance");
  }
}
```

Filled in by JSF before this action controller method is called.

The customer is not filled in automatically by JSF, since it is not directly part of the submitted data, but rather indirectly derived (by the business logic) from the submitted data. So, it is filled in by this action controller method.

There are five possible results pages: wrong-password.xhtml, unknown-customer.xhtml, negative-balance.xhtml, normal-balance.xhtml, and high-balance.xhtml. We are using the default mapping of return values to file names in all cases (rather than explicit navigation rules in faces-config.xml).

BankingBean.java: Part 3 (Placeholder for Results)

```
private Customer customer;

public Customer getCustomer() {
  return(customer);
}
```

Filled in by the action controller method based on the value returned by the business logic.

The getCustomer method is needed because the results page does #{bankingBean.customer.firstName} and #{bankingBean.customer.otherProperties}. But no setter method is needed since this property does not correspond directly to input data, and this property is not automatically filled in by JSF.

Business Logic (Interface)

```
public interface CustomerLookupService {
  public Customer findCustomer(String id);
}
```

Business Logic (Implementation)

```
public class CustomerSimpleMap
       implements CustomerLookupService {
  private Map<String,Customer> customers;

  public CustomerSimpleMap() {
    customers = new HashMap<String,Customer>();
    addCustomer(new Customer("id001", "Harry",
                            "Hacker", -3456.78));
    addCustomer(new Customer("id002", "Codie",
                            "Coder", 1234.56));
    addCustomer(new Customer("id003", "Polly",
                            "Programmer", 987654.32));
  }
```

Provides some simple hardcoded test cases.

Business Logic (Implementation, Continued)

```
  public Customer findCustomer(String id) {
    if (id != null) {
      return(customers.get(id.toLowerCase()));
    } else {
      return(null);
    }
  }

  private void addCustomer(Customer customer) {
    customers.put(customer.getId(), customer);
  }
}
```

Results Pages: Good Input (normal-balance.xhtml)

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html">
<h:head>
…
</h:head>
<h:body>
…
<ul>
  <li>First name: #{bankingBean.customer.firstName}</li>
  <li>Last name: #{bankingBean.customer.lastName}</li>
  <li>ID: #{bankingBean.customer.id}</li>
  <li>Balance: $#{bankingBean.customer.balanceNoSign}</li>
</ul>
…
</h:body></html>
```

negative-balance.xhtml and high-balance.xhtml are similar.

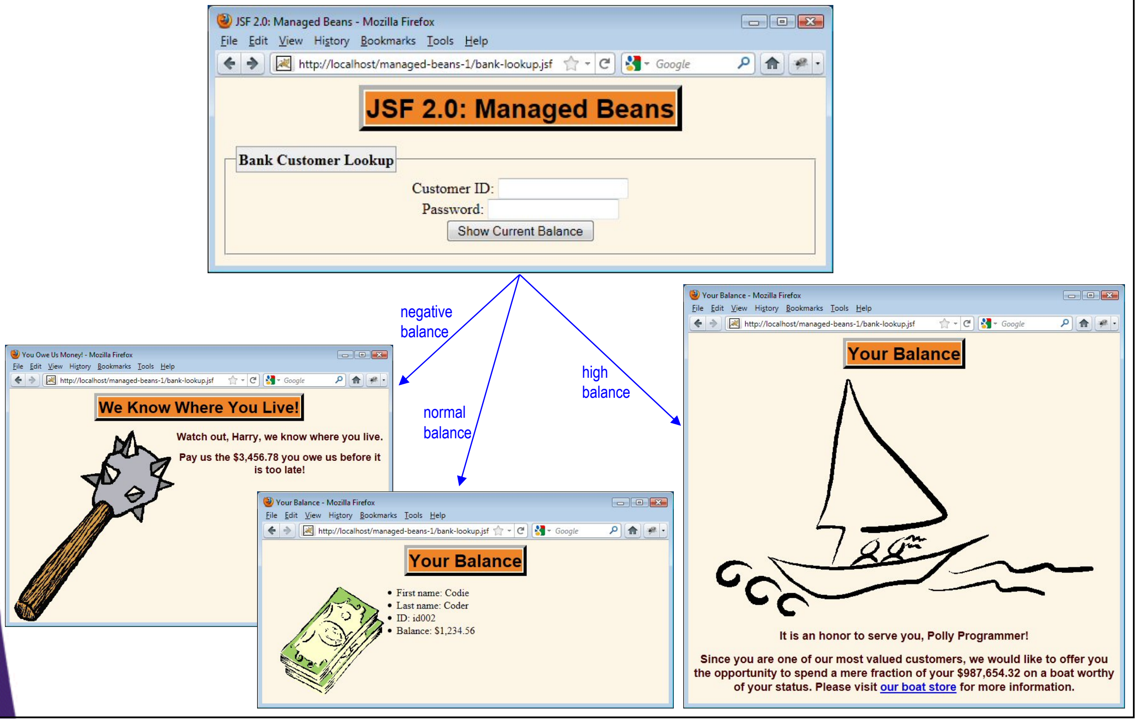
Results Pages: Bad Input (unknown-customer.xhtml)

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html">
<h:head>
…
</h:head>
<h:body>
…
<h2>No customer found with id "#{bankingBean.customerId}"</h2>
<p>Please <a href="bank-lookup.jsf">try again</a>.</p>
…
</h:body></html>
```

Even though customerId came from the user and could contain HTML tags, it is safe to use #{bankingBean.customerId} instead of <h:outputText value="#{bankingBean.customerId}"/>. The same HTML escaping is done for #{result} as for <h:outputText value="#{result}"/>

unknown-password.xhtml is similar.

Results (Legal ID and Password)


JSF 2.0: Managed Beans - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost/managed-beans-1/bank-lookup.jsf
Google
negative balance
normal balance
high balance
You Owe Us Money! - Mozilla Firefox
Your Balance - Mozilla Firefox
Show Current Balance

JSF 2.0: Managed Beans

Bank Customer Lookup

Customer ID:
Password:

We Know Where You Live!

Watch out, Harry, we know where you live.

Pay us the $3,456.78 you owe us before it is too late!

Your Balance

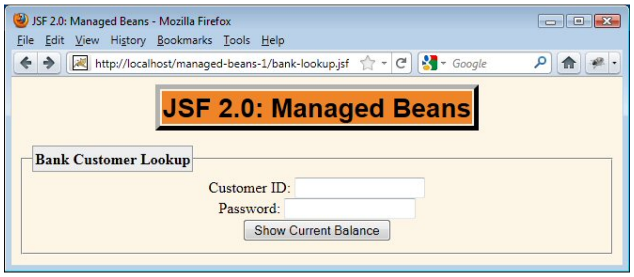
- First name: Codie
- Last name: Coder
- ID: id002
- Balance: $1,234.56

Your Balance

It is an honor to serve you, Polly Programmer!

Since you are one of our most valued customers, we would like to offer you the opportunity to spend a mere fraction of your $987,654.32 on a boat worthy of your status. Please visit our boat store for more information.

Results (Bad Inputs)


JSF 2.0: Managed Beans - Mozilla Firefox

http://localhost/managed-beans-1/bank-lookup.jsf

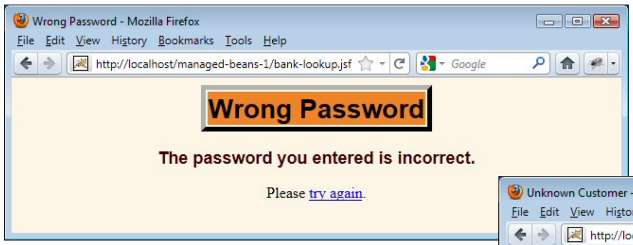
JSF 2.0: Managed Beans

Bank Customer Lookup

Customer ID:

Password:

Show Current Balance

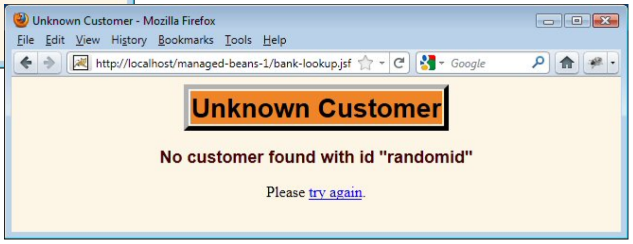

Wrong Password - Mozilla Firefox

http://localhost/managed-beans-1/bank-lookup.jsf

Wrong Password

The password you entered is incorrect.

Please try again.


Unknown Customer - Mozilla Firefox

http://localhost/managed-beans-1/bank-lookup.jsf

Unknown Customer

No customer found with id "randomid"

Please try again.

Prepopulating Input Fields

Dual Roles

- **Bean property refers to *both* getter & setter**
  - The getter method is called when form is displayed, and affects what is initially displayed to user
  - The value from the input element is passed to the setter method when the form is submitted
- **Examples**
  - <h:inputText…/>   (textfield)
    - If getter returns non-empty (neither null nor ""), this is initial value inside textfield. Not used for password fields.
  - <h:selectBooleanCheckbox…/> (checkbox)
    - Getter returns true: initially checked. Otherwise unchecked
  - <h:selectOneMenu…/> (combobox; drop down menu)
    - If return value of getter matches an entry in menu, that is initial selection. Otherwise top entry is initially selected.

Textfields

- **Example 1 (Simple property)**
  - <h:inputText value="#{someBean.someProperty}"/>
    - When form initially displayed, call <u>get</u>SomeProperty. If value is something other than null or empty String, use it as initial value of textfield.
    - When form submitted, take the value in the textfield and pass it to <u>set</u>SomeProperty.
- **Example 2 (Chained properties)**
  - <h:inputText value="#{someBean.a.b.c}"/>
    - When form initially displayed, call getA, then getB on that result, then getC on that. If value of getC is non-empty, use it as initial value of textfield.
    - When form submitted, call <u>get</u>A, then <u>get</u>B on that result. Then take the value in the textfield and pass it to the <u>set</u>C method of that final result. Only last one becomes setter.

Checkboxes

- **Example**
  - <h:selectBooleanCheckbox
    value="#{someBean.someProperty}"/>
    - When form initially displayed, call the accessor method, which must be of type boolean or Boolean. The name could be either isSomeProperty (most common) or getSomeProperty. If value is true, make checkbox initially checked. If value is false, make checkbox initially unchecked.
    - When form submitted, pass either true or false to setSomeProperty (which expects boolean or Boolean)
      - Note that JSF does not require anything (such as Struts 1.*x* does) to clear out old values of session-scoped checkboxes or radio buttons.

Drop Down Menus (Comboboxes)

- **Example**
  - <h:selectOneMenu
    value="#{someBean.someProperty}">
    <f:selectItems value="#{someBean.choices}"/>
    </h:selectOneMenu>
    - When form initially displayed, call getChoices (which should return List<SelectItem> or SelectItem[]). This gives the entries in the drop down menu. Then call getSomeProperty. If result matches any of the entries, make that initially shown item. Otherwise use top item.
      - There are several forms of the SelectItem constructor, but the simplest just takes a String: the value that will go in the menu.
    - When form submitted, pass the current selection to setSomeProperty.
      - Note that, due to the f:selectItems tag, you must add an entry for the f: namespace to the <html …> start tag at top of file

Example

- **Idea**
  - Collect input about programming background and preferences. Give recommended computer study plan.
- **Input form**
  - Textfield for favorite language. Prepopulate based on most popular language in the application
  - Drop down menu for second-favorite language. Make choices be the languages for which app has study guides. Make initial choice the second most popular language in the application.
  - Two checkboxes. Initial selection based on logic in app
- **Results page**
  - List of recommended languages. Requires looping tag.
    - Shown briefly here but covered in detail in later section

Input Form – Top (study-plan-input.xhtml)

```
<!DOCTYPE …>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:h="http://java.sun.com/jsf/html">
<h:head>
…
</h:head>
```

Since we use the f:selectItems tag for the drop down menu, we must declare the f: namespace.

Input Form – Bottom (study-plan-input.xhtml)

```
<h:body>
…
<h:form>
  Email address:
  <h:inputText value="#{trainingForm.emailAddress}"/><br/>
  Favorite language:
  <h:inputText value="#{trainingForm.favoriteLanguage}"/><br/>
  Second favorite language:
  <h:selectOneMenu value="#{trainingForm.secondFavoriteLanguage}">
    <f:selectItems value="#{trainingForm.availableLanguages}"/>
  </h:selectOneMenu><br/>
  Programmed for 5+ years?
  <h:selectBooleanCheckbox value="#{trainingForm.expert}"/><br/>
  Personally know Larry Page, Steve Balmer, and James Gosling?
  <h:selectBooleanCheckbox value="#{trainingForm.liar}"/><br/>
  <h:commandButton value="Show Recommended Study Plan"
                   action="#{trainingForm.showTrainingPlan}"/>
</h:form>
…
</h:body></html>
```

Since getFavoriteLanguage returns non-empty (Java) originally, this textfield has an initial value when form displayed to user.

Since getSecondFavoriteLanguage returns a value that matches one of the choices in the list of languages, that option (JavaScript) is initially displayed to the user, rather than the top item being initially displayed.

isExpert initially returns true, so the first checkbox is initially checked.
isLiar initially returns false, so the second checkbox is initially unchecked.

Managed Bean (Part 1 – Properties for Input Elements)

```
@ManagedBean
public class TrainingForm {
  private String emailAddress;
  private String favoriteLanguage =
    LanguageUtils.findMostPopularLanguage(0);
  private String secondFavoriteLanguage =
    LanguageUtils.findMostPopularLanguage(1);
  private boolean isExpert = true;
  private boolean isLiar = false;

  // Getters and setters for all. The getters for
  // the boolean properties start with is, not get

  public List<SelectItem> getAvailableLanguages() {
    return(LanguageUtils.languageList());
  }  // Options for dropdown box. See later slide.
```

Managed Bean (Part 2 – Action Controller)

```
public String showTrainingPlan() {
  int numLanguagesToStudy;
  if (isExpert) {
    numLanguagesToStudy = 4;
  } else {
    numLanguagesToStudy = 2;
  }
  if (isLiar) {
    return("liar");
  } else {
    languagesToStudy =
      LanguageUtils.randomLanguages(numLanguagesToStudy);
    return("study-plan");
  }
}
```

This List is the placeholder that is filled in. Since it can be of varying length, the results page will need to use a loop.

Managed Bean (Part 3 – Placeholder for Results)

```
private List<String> languagesToStudy;

public List<String> getLanguagesToStudy() {
  return(languagesToStudy);
}
```

Helper Class (Code for Options in Dropdown)

```
public class LanguageUtils {
  private static String[] languages =
    { "Java", "JavaScript", "C#", "C++", "PHP", "Python",
      "Perl", "Ruby", "Scala" };
  private static List<SelectItem> availableLanguages;

  static {
    availableLanguages = new ArrayList<SelectItem>();
    for(String language: languages) {
      availableLanguages.add(new SelectItem(language));
    }
  }

  public static List<SelectItem> languageList() {
    return(availableLanguages);
  }
…
```

Main Results Page (study-plan.xhtml)

```
<!DOCTYPE …>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:ui="http://java.sun.com/jsf/facelets">
<h:head>…</h:head>
<h:body>
…
<ul>
  <ui:repeat var="language"
             value="#{trainingForm.languagesToStudy}">
    <li>#{language}</li>
  </ui:repeat>
</ul>
…
</h:body></html>
```

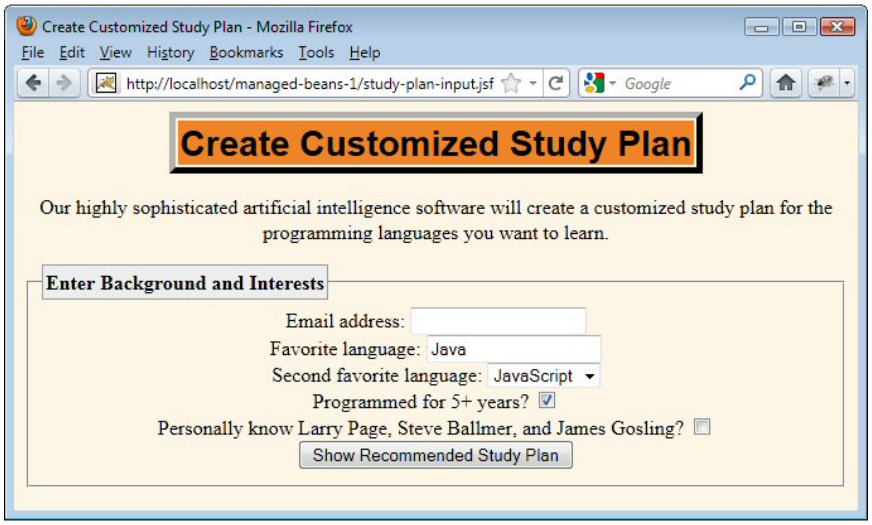
There is a whole separate tutorial section on looping and handling variable-length data. But the basic idea is simple: identical to the JSTL c:forEach loop and very similar to the Java for(Blah b: collectionOfBlahs) loop. This uses JSTL behind the scenes, so if you are using a non-Java-EE-6 server, you must include the JSTL JAR files with your app.

“Error” Page (liar.xhtml)

```
<!DOCTYPE …>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html">
<h:head>…</h:head>
<h:body>
…
<h2>Yeah, right. And you know more JavaScript than
Brendan Eich or Doug Crockford, I suppose?</h2>
<p>Please <a href="study-plan-input.jsf">try again</a>,
but be honest this time.</p>
…
</h:body></html>
```

Insulting your users is not generally considered a recommended practice.

Results (Input Form)


Create Customized Study Plan - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost/managed-beans-1/study-plan-input.jsf
Google

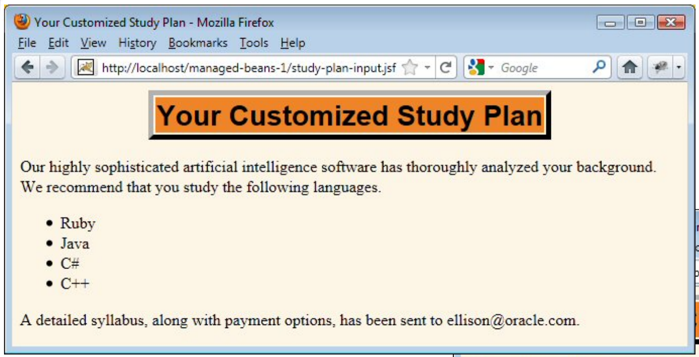
Create Customized Study Plan

Our highly sophisticated artificial intelligence software will create a customized study plan for the programming languages you want to learn.

Enter Background and Interests

Email address:
Favorite language: Java
Second favorite language: JavaScript
Programmed for 5+ years? ☑
Personally know Larry Page, Steve Ballmer, and James Gosling? ☐
Show Recommended Study Plan

Results (Results Pages)

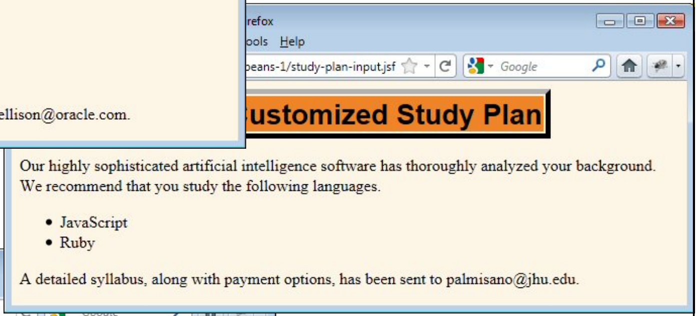

Your Customized Study Plan - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost/managed-beans-1/study-plan-input.jsf

Your Customized Study Plan

Our highly sophisticated artificial intelligence software has thoroughly analyzed your background. We recommend that you study the following languages.

- Ruby
- Java
- C#
- C++

A detailed syllabus, along with payment options, has been sent to ellison@oracle.com.

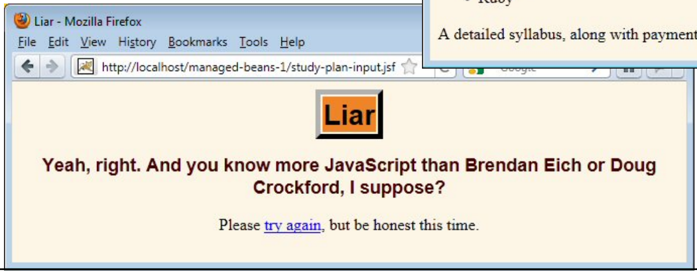


ustomized Study Plan

Our highly sophisticated artificial intelligence software has thoroughly analyzed your background. We recommend that you study the following languages.

- JavaScript
- Ruby

A detailed syllabus, along with payment options, has been sent to palmisano@jhu.edu.


Liar - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost/managed-beans-1/study-plan-input.jsf

Liar

Yeah, right. And you know more JavaScript than Brendan Eich or Doug Crockford, I suppose?

Please try again, but be honest this time.

Wrap-Up

Summary

- **Managed beans generally have three sections**
  - Bean properties to represent input elements
  - Action controller method
  - Placeholder for results (properties derived from input)
- **Prepopulating input elements**
  - <h:inputText value="#{customer.firstName}"/>
    - When form displayed, getFirstName called
    - When form submitted, value passed to setFirstName
- **Drop down menus (combo boxes)**

```
<h:selectOneMenu value="#{bean.userChoice}">
   <f:selectItems value="#{bean.listOfOptions}"/>
</h:selectOneMenu>
```

- **Looping tags**
  - Look at simple example, but covered in detail in later lecture

Questions?