Managed Beans I –
Classes to Represent Form Info

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

Customized Java EE Training: http://courses.coreservlets.com/
Java 7, Java 8, JSF 2, PrimeFaces, Android, JSP, Ajax, jQuery, Spring MVC, RESTful Web Services, GWT, Hadoop
Developed and taught by well-known author and developer. At public venues or onsite at your location.

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

- **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)
- **Business logic**
  - How to prevent changes in the way that the data is found from rippling through the rest of the code
    - Making separate method
    - Passing and returning simple types
    - Coding to interfaces
    - Using dependency injection

Customized Java EE Training: http://courses.coreservlets.com/
Java 7, Java 8, JSF 2, PrimeFaces, Android, JSP, Ajax, jQuery, Spring MVC, RESTful Web Services, GWT, Hadoop
Developed and taught by well-known author and developer. At public venues or onsite at your location.
Background: Basic JavaBeans

• **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 `getBlah` and `setBlah`
    • 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 `isBlah` instead of `getBlah`
    • 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. Instance variable 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` (`isPrime` is preferred)
    • 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}`
Bean Properties: Examples

<table>
<thead>
<tr>
<th>Method Names</th>
<th>Property Name</th>
<th>Example JSF Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>getFirstName</td>
<td>firstName</td>
<td># {customer.firstName}</td>
</tr>
<tr>
<td>setFirstName</td>
<td></td>
<td>&lt;h:inputText value=&quot;# {customer.firstName}&quot;/&gt;</td>
</tr>
<tr>
<td>isExecutive</td>
<td>executive</td>
<td># {customer.executive}</td>
</tr>
<tr>
<td>setExecutive</td>
<td></td>
<td>&lt;h:selectBooleanCheckbox value=&quot;# {customer.executive}&quot;/&gt;</td>
</tr>
<tr>
<td>getExecutive</td>
<td>executive</td>
<td># {customer.executive}</td>
</tr>
<tr>
<td>setExecutive</td>
<td></td>
<td>&lt;h:selectBooleanCheckbox value=&quot;# {customer.executive}&quot;/&gt;</td>
</tr>
<tr>
<td>getZIP</td>
<td>ZIP</td>
<td># {address.ZIP}</td>
</tr>
<tr>
<td>setZIP</td>
<td></td>
<td>&lt;h:inputText value=&quot;# {address.ZIP}&quot;/&gt;</td>
</tr>
</tbody>
</table>

Note 1: property name does not exist anywhere in your code. It is just a shortcut for the method names. Instance variable name is irrelevant.

Note 2: if you can choose the method names, it is better 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
      ```java
      public double speed;
      ```
    - Right
      ```java
      private double speed; // Var name need not match method name
      ```
      ```java
      public double getSpeed() {
        return(speed);
      }
      ```
      ```java
      public void setSpeed(double speed) {
        this.speed = speed;
      }
      ```

- **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

```java
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.

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

```java
// 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

```java
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}
    - getHTML method: refer to “HTML” (not “hTML”)
      - #{message.HTML}
Managed Beans

• **JSF automatically “manages” certain beans**
  – Instantiates it
    • Thus the need for a zero-arg constructor
  – Controls its lifecycle
    • Scope (request, session, application, etc.) 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 managed beans**
  – Simplest: `@ManagedBean` before class
    • Results in request scope. See next lecture for other scopes.
  – More 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, depending on JSF version. Details:
        » [http://www.java.net/node/706733](http://www.java.net/node/706733)
  – If getter method talks to database or does other expensive operation, performance can be unexpectedly bad.

• **Solution**
  – Have action controller store data in instance variables, have getter methods merely return the existing values.
    • Applies to managed beans only, not to “regular” beans.
Business Logic and 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 no requirement to have a setter method.
JSF Flow of Control
(Updated but Still Simplified)

balance.xhtml
Uses <h:commandButton ... action="# \{bankingBean.showBalance\}" and <h:inputText value="# \{bankingBean.customerId\}"/>

When form first displayed, bankingBean is instantiated and getCustomerId is called. If result is non-empty, it becomes initial value of the textfield.

As submitted, textfield value passed to setCustomerId.

Business Logic

• Big idea
  – The results page usually shows more than just the data the user entered: it normally also shows data derived from the user data (e.g., the user enters a bank account number and you display the account balance).

• Goal
  – Isolate the code that looks up the derived data, so that changes to the way the data is calculated do not require changes in the rest of the code. See next slide for four approaches. The first two (separate methods and simple types) should always be done and the other two (coding to interfaces and using dependency injection) should be considered for complex applications.

Business Logic: Overview
Approaches to Business Logic: Summary

• **Use a separate method** (do always)
  – Do not compute the derived data directly in the action controller method, but use a separate method.

• **Simple types in, simple types out** (do always)
  – Never return a ResultSet or anything specific to how you found the data. Return an object representing the result itself.

• **Code to interfaces** (do usually)
  – Make an interface such as CustomerLookupService and use that type. Prevents accidental dependence on concrete type.

• **Use dependency injection** (do sometimes)
  – Inject the concrete type so nothing in main class changes when you swap out concrete implementations of the interface.

 Interfaces and Dependency Injection: Analogous Example

• **List<String> names = new ArrayList<>();**

• **Questions**
  – What is benefit of using List instead of ArrayList above?
  – What if you want to switch from ArrayList to LinkedList without changing any code in the main class?
Approaches to Business Logic: Details

- **Use a separate method**
  - Do not compute the derived data directly in the action controller method, but use a separate method. E.g.:
    ```java
    private static CustomerLookupService lookupService = …;
    ...
    Customer cust = lookupService.findCustomer(idFromUser);
    ```

- **Simple types in, simple types out**
  - Never return a ResultSet, a Spring object, a Hibernate object, a Web Services object, or anything specific to how you found the data. Return a Java object representing the results data itself.
  - E.g., Customer above is a POJO that represents the final answer; it has no ties to the specific manner in which the customer was found from the ID.

Approaches to Business Logic: Details (Continued)

- **Code to interfaces**
  - Make an interface such as CustomerLookupService and use that type so you cannot accidentally do something specific to the concrete implementation.
    ```java
    private static CustomerLookupService lookupService =
    new SomeConcreteVersion();
    ```

- **Use dependency injection**
  - In the above example, there is still one line of code that has to change when you switch implementations. Inject the concrete type so nothing in main class changes when you switch concrete implementations of the interface. Change separate class (the bean named currentLookupService) instead.
    ```java
    @ManagedProperty(value="#{currentLookupService}")
    private CustomerLookupService lookupService;
    ```
    - Details on this syntax covered in next section (Managed Beans 2)
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)

```xml
<!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://xmlns.jcp.org/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 re-instantiated (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).
private Customer customer;

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

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.

public interface CustomerLookupService {
    public Customer findCustomer(String id);
}
public class CustomerSimpleMap
    implements CustomerLookupService {
    private Map<String, Customer> customers;

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

    @Override
    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)

```html
<!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://xmlns.jcp.org/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>
```

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

```html
<!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://xmlns.jcp.org/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>
```
Results (Legal ID and Password)

- JSF 2: Managed Beans
- We Know Where You Live!
- Watch out, Harry, we know where you live. Pay us the $3,456.78 you owe us before we repossess your house!
- 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)

- Wrong Password
  - The password you entered is incorrect.
  - Unknown Customer
    - No customer found with id "bogusid"
    - Please try again
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)

• Business logic
  – Apply strategies to limit ripple effect when data-lookup methods change. Most importantly, never return something like ResultSet or Hibernate object specific to the way in which you found the answer. Instead, return simple Java object representing the answer itself.
Questions?

More info:
http://www.coreservlets.com/jsf-tutorial/jsf2/ – JSF 2.2 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

Customized Java EE Training: http://courses.coreservlets.com/
Java 7, Java 8, JSF 2, PrimeFaces, Android, JSP, Ajax, jQuery, Hadoop, RESTful Web Services, GWT, Hadoop
Developed and taught by well-known author and developer. At public venues or onsite at your location.