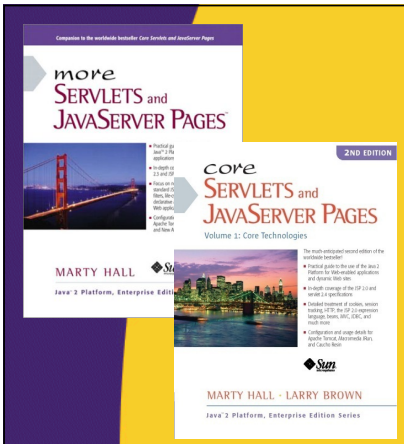
Managed Beans II – Advanced Features

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



Topics in This Section

- **Custom bean names**
- **Bean scopes**
  - Especially session-scoped beans
- **Getting the "raw" request and response objects**
- **Dependency injection**

Giving Custom Names to Beans

Main Point

- **The name attribute of @ManagedBean**

```
@ManagedBean(name="anyName")
public class BeanName { … }
```

  - You refer to bean with #{anyName.blah}, where bean name is exact value of name attribute.
    - Still request scoped by default.
    - Still no entries in faces-config.xml

Example

- **Idea (same behavior as Navigator example)**
  - Click on button in initial page
  - Get one of three results pages, chosen at random
- **What you need (same except for bean name)**
  - A starting page

```
<h:commandButton…action="#{customName.choosePage}"/>
```

  - A bean
    - Name: Navigator2 (not related to bean name above)
    - @ManagedBean(name="customName")
    - choosePage returns 3 possible Strings as before
      - "page1", "page2", or "page3"
  - Three results pages as before
    - Names matching method return values
      - page1.xhtml, page2.xhtml, and page3.xhtml

start-page2.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>
…
<fieldset>
<legend>Random Results Page</legend>
<h:form>
  Press button to get one of three possible results pages.
  <br/>
  <h:commandButton value="Go to Random Page"
                   action="#{customName.choosePage}"/>
</h:form>
</fieldset>
…
</h:body></html>
```

Except for bean name, same as start-page.xhtml from the Programming Basics lecture.

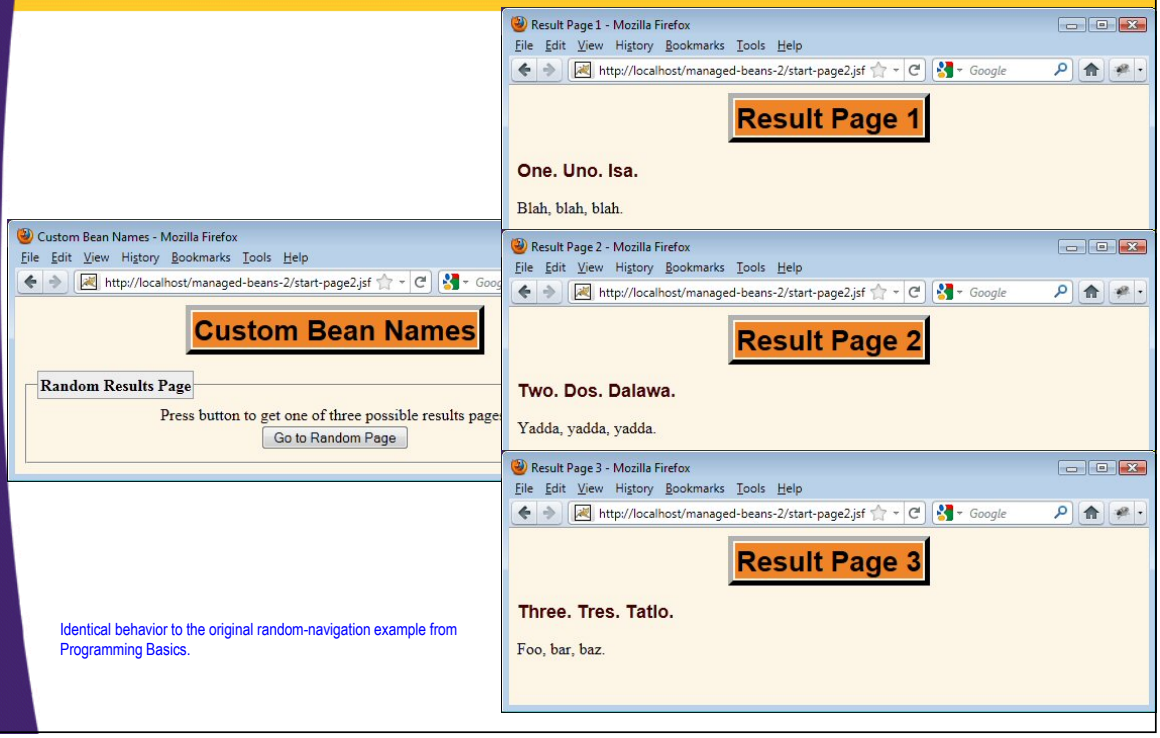
Navigator2.java

```
package coreservlets;

import javax.faces.bean.*;

@ManagedBean(name="customName")
public class Navigator2 extends Navigator {}
```

Uses the name “customName” instead of “navigator2”.

Inherits the choosePage method that randomly returns "page1", "page2", or "page3". Since in this simple example the results pages have only static text (no use of beans), this example shares the results pages with the original Navigator example: page1.xhtml, page2.xhtml, and page3.xhtml.

Results



Identical behavior to the original random-navigation example from Programming Basics.

Controlling Bean Scopes

Bean Scopes

- **Idea**
  - Designates how long managed beans will stay "alive", and which users and requests can access previous bean instances.
- **JSF 1.*x* scopes**
  - Request, session, application
  - Specified in faces-config.xml
  - Request scope is the default
- **JSF 2.0 scopes**
  - request, session, application, view, none, custom
  - Specified either in faces-config.xml or by one of the new annotations (e.g., @SessionScoped)
  - Request scope is still the default

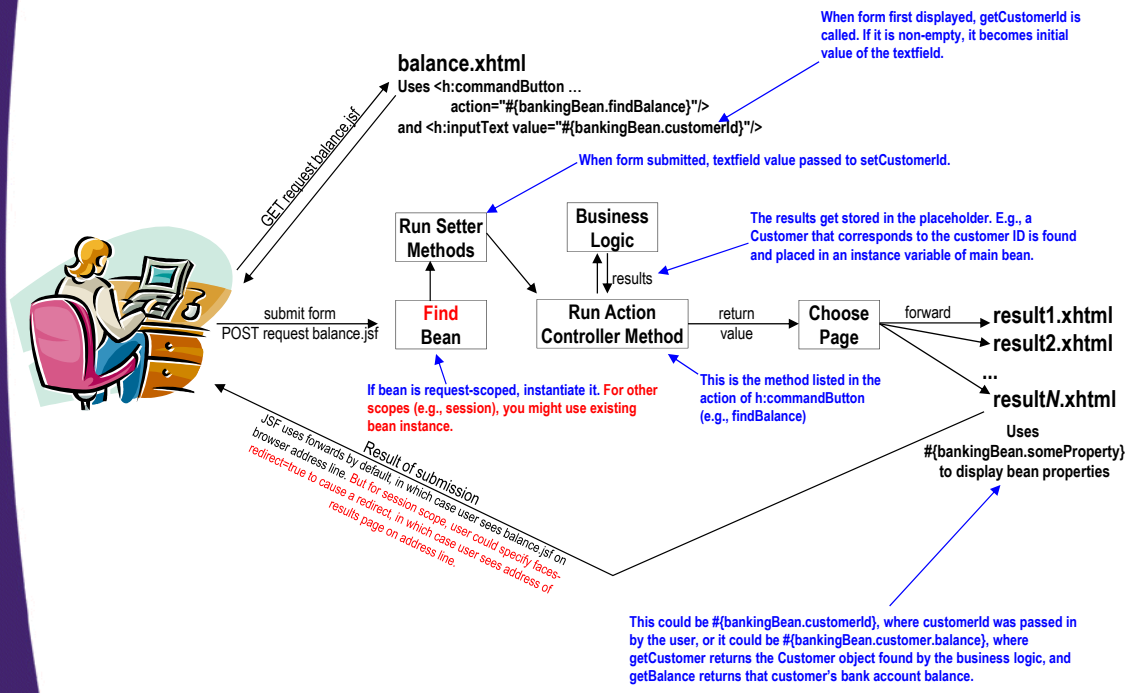
Annotations to Specify Bean Scope

- **@RequestScoped**
  - Default. Make a new instance for every HTTP request. Since beans are also used for initial values in input form, this means bean is generally instantiated twice (once when form is displayed, once when form is submitted).
- **@SessionScoped**
  - Put bean in session scope. If same user with same cookie returns before session timeout, same bean instance is used. You should make bean Serializable.
- **@ApplicationScoped**
  - Put bean in application scope. Shared by all users. Bean either should have no mutable state or you must carefully synchronize access.

Annotations to Specify Bean Scope (Continued)

- **@ViewScoped**
  - Same bean instance is used as long as same user is on same page (e.g., with event handlers or Ajax).
    - New scope in JSF 2.0.
    - Bean should implement Serializable
- **@CustomScoped(value="#{someMap}")**
  - Bean is stored in the Map, and programmer can control lifecycle.
    - New scope in JSF 2.0
- **@NoneScoped**
  - Bean is not placed in a scope. Useful for beans that are referenced by other beans that *are* in scopes.
    - New scope in JSF 2.0

JSF Flow of Control (Updated but Still Simplified)



Main Points

- **You can use annotations to give scope**
  - @RequestScoped (same as omitting scope)
  - @SessionScoped
  - @ApplicationScoped
  - @ViewScoped
  - @CustomScoped
  - @NoneScoped

  - Traditionally placed after @ManagedBean

```
@ManagedBean
@SessionScoped
public class SomePojo { … }
```


Application Scope: When Used

- **For beans that have no changeable state**
  - Navigation rules only (no getters/setters)
    - As in Navigator example earlier (this is rare in real life)
  - Supplies list of choices for drop down menus

```
<h:selectOneMenu value="#{requestScopedBean.choice}">
    <f:selectItems value="#{appScopedBean.options}"/>
</h:selectOneMenu>
```

  - Data structures used as properties in main beans
    - See later section on @ManagedBeanProperty
- **For beans you deliberately share**
  - Shared across all users and all pages
  - You have to very carefully use synchronization to avoid race conditions
    - Quite rare in real life

Application Scope: Syntax

- **Usual**

  @ManagedBean
  @ApplicationScoped
  public class SomeClassNoUserState { … }
  - Class is instantiated first time it is used. After that, all users and all requests share the same instance.
- **Occasional**

  @ManagedBean(eager=true)
  @ApplicationScoped
  public class SomeClassBigDataNoUserState { … }
  - Class is instantiated when the app is loaded (which is usually when the server starts). After that, all users and all requests share the same instance. Useful if the class has a big data structure that takes a long time to initialize.

Example: Bean with No Changeable State

- **Navigator from Programming Basics lecture**
  - There was no state (no instance variables), just an action controller method
  - So, application scope prevents unnecessary object instantiation.
    - In practice, instantiation of small objects is very fast, so this "optimization" is probably not even measurable.
      - And causes problems if you later go and add input fields and corresponding accessor methods. So, for Navigator example, the benefit of application scope is questionable at best.
    - For large data structures (especially large Maps), this trick can help significantly, though.
      - Large list of choices for drop down menus
      - Maps for business logic
        - » In both cases above, previous examples used static variables to work around this problem. Application scope is a reasonable alternative.

Navigator.java

```
package coreservlets;

import javax.faces.bean.*;

@ManagedBean
@ApplicationScoped
public class Navigator {
  private String[] resultPages =
    { "page1", "page2", "page3" };

  public String choosePage() {
    return(RandomUtils.randomElement(resultPages));
  }
}
```

Since there is no mutable state, all users can share the same instance of this bean. For this example, using application scope only saves recreating the tiny array, so has little-to-no measurable performance benefit, and it causes huge problems later if you add accessor methods for storing user-specific data, but forget to remove @ApplicationScoped. Still, if the data structure were large and you had no mutable state, this trick would buy you something.

The xhtml pages and results were shown in the Programming Basics lecture. One of page1.xhtml, page2.xhtml, and page3.xhtml is displayed at random.

Session Scope: Main Points

- **Bean instance reused if**
  - Same user
  - Same browser session
    - Usually based on cookies, but can be based on URL rewriting
- **Useful for**
  - Remembering user preferences
  - Prefilling values from previous entries
  - Accumulating lists of user data (ala shopping carts)
- **Normal Java session tracking rules apply**
  - Custom classes should be Serializable
    - Some servers save session data to disk on restart
    - Distributed Web apps need this to replicate sessions

Session Scope: Example

- **Idea**
  - Small variation of banking example
  - Remember previously entered id
    - If end user comes back to input page, the ID they entered last time is already filled in
- **What you need**
  - Add @SessionScoped for bean
  - Make bean Serializable
  - Optionally, add faces-redirect=true to end of return values, to tell JSF to redirect (instead of forward) to results pages
    - Allows users to access results pages directly

bank-lookup2.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 (Session Scope)</legend>
<h:form>
  Customer ID:
  <h:inputText value="#{bankingBean2.customerId}"/><br/>
  Password:
  <h:inputSecret value="#{bankingBean2.password}"/><br/>
  <h:commandButton value="Show Current Balance"
                   action="#{bankingBean2.showBalance}"/>
</h:form>
</fieldset>
…
</h:body></html>
```

Same as bank-lookup.xhtml except for bean name and some minor changes to text.

BankingBean2.java

```
import java.io.*;
import javax.faces.bean.*;

@ManagedBean
@SessionScoped
public class BankingBean2 extends BankingBean
                          implements Serializable {
  @Override
  public String showBalance() {
    String origResult = super.showBalance();
    return(origResult + "2?faces-redirect=true");
  }
}
```

If a page uses the name bankingBean2 and is accessed by the same user in the same browser session, the same bean instance will be used.

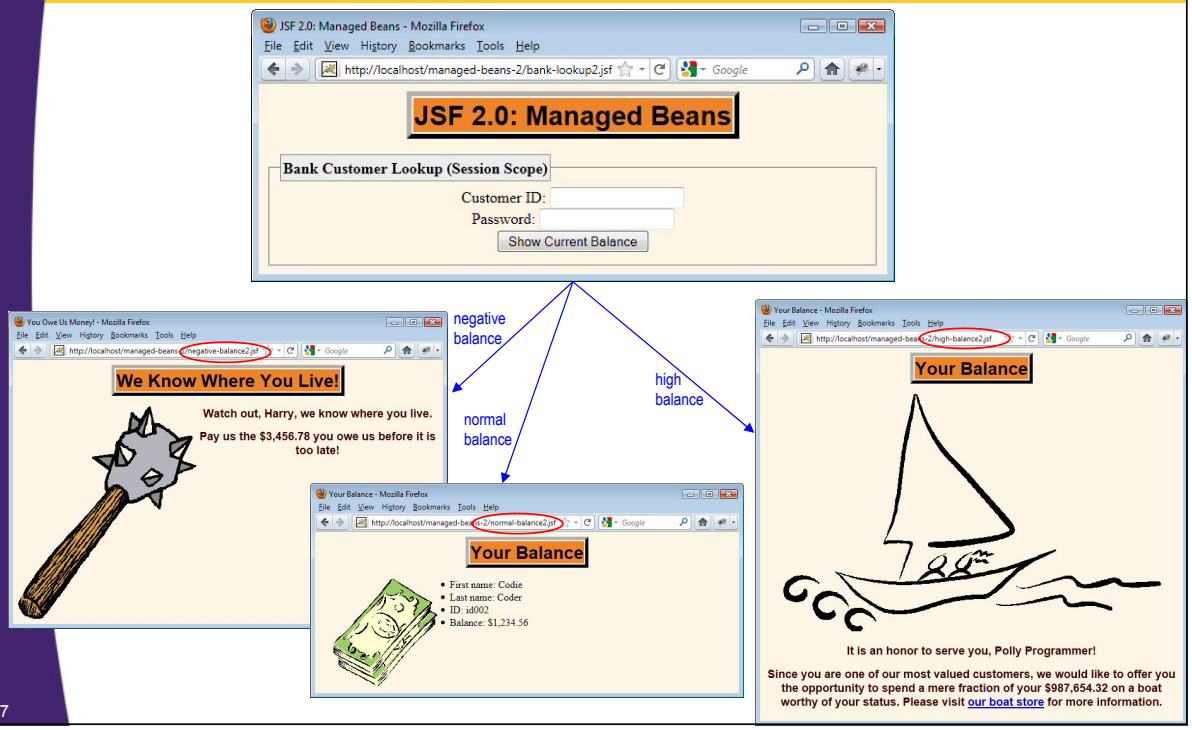
Results pages are negative-balance2.xhtml, normal-balance2.xhtml, etc. By also appending faces-redirect=true, JSF will redirect instead of forward to the results pages, thus exposing the URLs of the results pages and letting users navigate directly to them later.

normal-balance2.xhtml

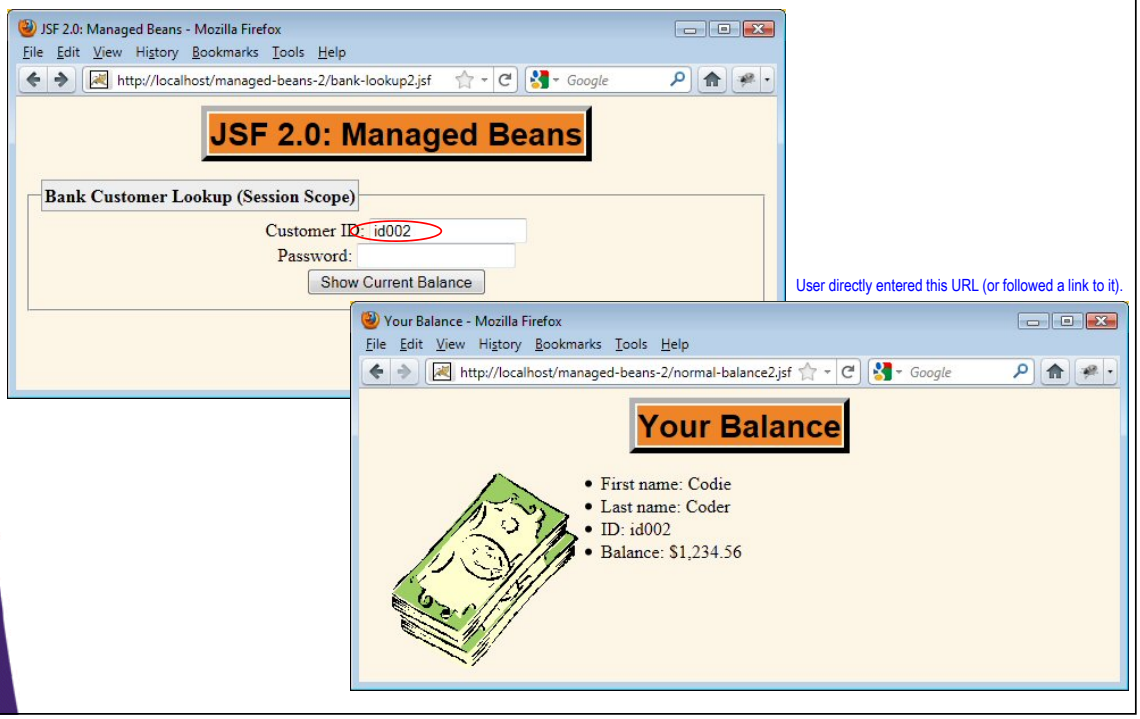
```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html">
<h:head>
…
</h:head>
<h:body>
…
<ul>
  <li>First name: #{bankingBean2.customer.firstName}</li>
  <li>Last name: #{bankingBean2.customer.lastName}</li>
  <li>ID: #{bankingBean2.customer.id}</li>
  <li>Balance: $#{bankingBean2.customer.balanceNoSign}</li>
</ul>
…
</h:body></html>
```

Same as normal-balance.xhtml except for bean name.
negative-balance2.xhtml and high-balance2.xhtml are similar.

Results (Initial Access)



Results (User Returns in Same Browser Session)



Getting the Request and Response Objects

Issue

- **No automatic access to request & response**
  - JSF action controller methods do not have direct access
    - Unlike in Struts, where action controller method (execute) gets request and response automatically
- **Good news**
  - In most cases, only use for request and response objects is for explicit user data (request parameters), and JSF provides a much simpler way to get them.
  - Having your form beans be POJOs is very convenient
- **Bad news**
  - In the cases where you need the request and response objects, code is more awkward
  - JSF programmers forget how valuable the request and response objects are

Why You Need Request and Response Objects

- **Uses for request object**
  - Explicit session manipulation
    - E.g., changing inactive interval or invalidating session
  - Explicit cookie manipulation (e.g., long-lived cookies)
  - Reading request headers (e.g., User-Agent)
  - Looking up requesting host name
- **Uses for response object**
  - Setting status codes
  - Setting response headers
  - Setting long-lived cookies

Solution

- **Static methods**
  - If they are needed, use static method calls to get them

```
ExternalContext context =
   FacesContext.getCurrentInstance().getExternalContext();
HttpServletRequest request =
   (HttpServletRequest)context.getRequest();
HttpServletResponse response =
   (HttpServletResponse)context.getResponse();
```

- **Note**
  - In some environments, you cast results of getRequest and getResponse to values other than HttpServletRequest and HttpServletResponse
    - E.g., in a portlet environment, you might cast result to PortletRequest and PortletResponse

Example

- **Idea**
  - Collect a search string and a search engine name, show the results of a search with that search engine.
- **Input form**
  - Use textfield for arbitrary search string. Use drop down menu to list only search engines that app supports.
- **Managed bean**
  - Construct a URL by concatenating a base URL (e.g., http://www.google.com/search?q=) with the URL-encoded search string
  - Do response.sendRedirect
    - Must use static methods to get the HttpServletResponse
  - Return normal strings for error pages

Input Form (search-engine-form.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>
<h:body>
…
<h:form>
  Search String:
  <h:inputText value="#{searchController.searchString}"/><br/>
  Search Engine:
  <h:selectOneMenu value="#{searchController.searchEngine}">
    <f:selectItems value="#{searchController.searchEngineNames}"/>
  </h:selectOneMenu><br/>
  <h:commandButton value="Search"
                   action="#{searchController.doSearch}"/>
</h:form>
</h:body></html>
```

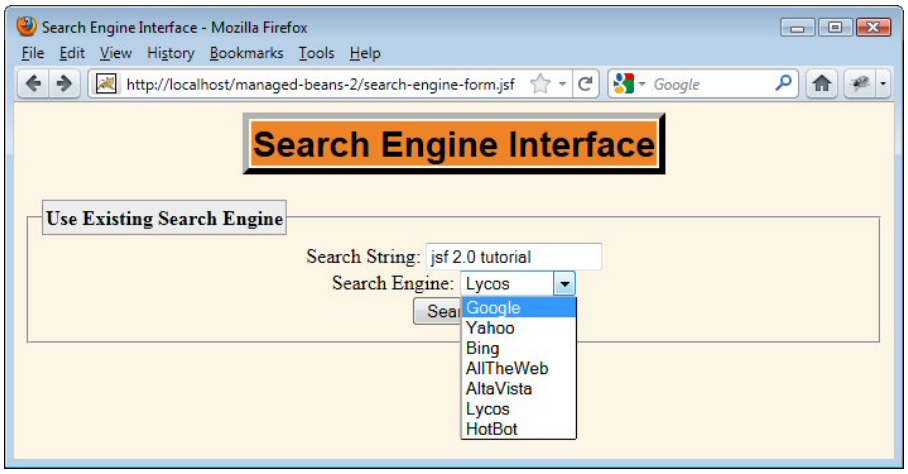
Managed Bean (Part 1 – Properties for Input Elements)

```
@ManagedBean
public class SearchController {
  private String searchString="", searchEngine;

  public String getSearchString() {
    return(searchString);
  }
  public void setSearchString(String searchString) {
    this.searchString = searchString.trim();
  }
  public String getSearchEngine() {
    return(searchEngine);
  }
  public void setSearchEngine(String searchEngine) {
    this.searchEngine = searchEngine;
  }
  public List<SelectItem> getSearchEngineNames() {
    return(SearchUtilities.searchEngineNames());
  }
```

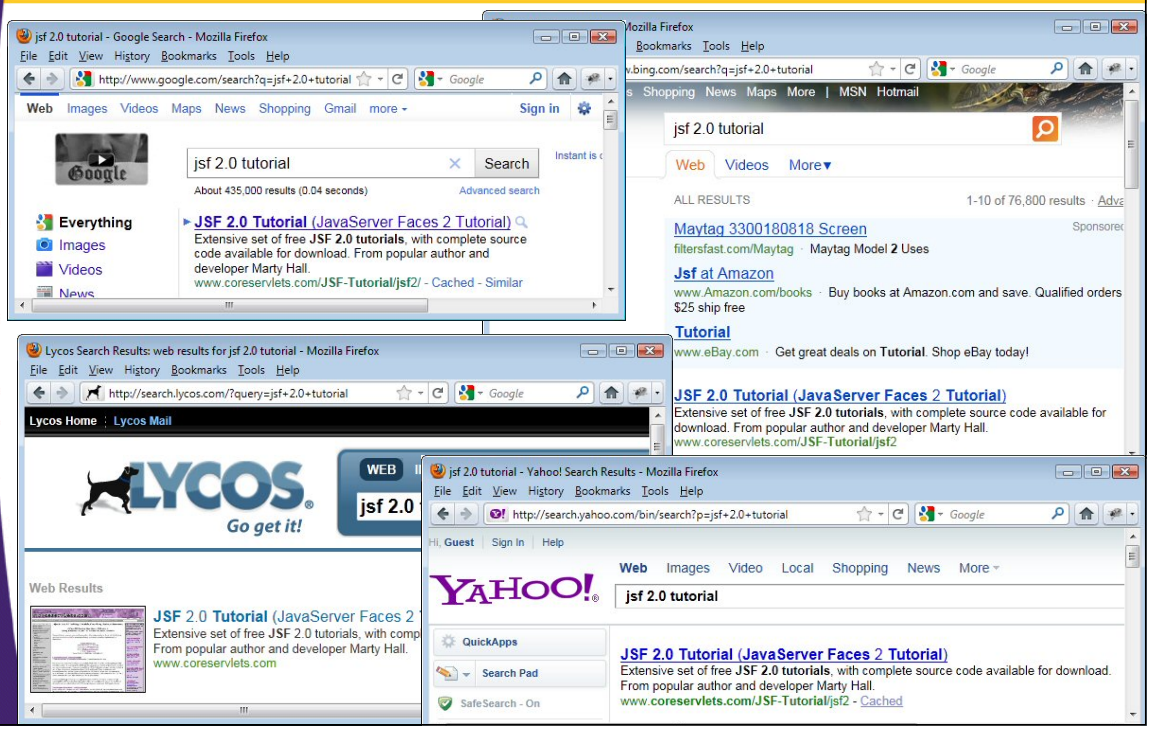

Managed Bean (Part 2 – Action Controller)

```
  public String doSearch() throws IOException {
    if (searchString.isEmpty()) {
      return("no-search-string");
    }
    searchString = URLEncoder.encode(searchString, "utf-8");
    String searchURL =
      SearchUtilities.makeURL(searchEngine, searchString);
    if (searchURL != null) {
      ExternalContext context =
        FacesContext.getCurrentInstance().getExternalContext();
      HttpServletResponse response =
        (HttpServletResponse)context.getResponse();
      response.sendRedirect(searchURL);
      return(null);
    } else {
      return("unknown-search-engine");
    }
  }
```

Results (Input Form)



Results (Forwarding Results)



Advanced Topic: Using @ManagedProperty

@ManagedProperty: Main Points

- **JSF supports simple dependency injection**
  - That is, you can assign values to a managed bean property (i.e., a value the main bean depends on) without hardcoding it into the class definition
    - Not as powerful as with Spring, but still useful
  - @ManagedProperty lets you do this with annotations
    - @ManagedProperty(value="#{someBean}")
      private SomeType someField;
  - <managed-property> lets you do it in faces-config.xml
    - This is same as in JSF 1.*x*
- **Setter method for the property is required**
  - E.g., in example above, you must have setSomeField
    - You can use “name” attribute of @ManagedProperty if setter method name does not match field name

@ManagedProperty: Secondary Points

- **You can instantiate beans at app load time**
  - @ManagedBean(eager=true)
    @ApplicationScoped
    public class SomePojo { … }
    - This is useful for the bean that you inject into the main bean. The injected bean is often something shared like a lookup service, whereas the main bean usually contains user-specific data
- **faces-config better than annotations (?)**
  - One of the points of dependency injection is to let you change the concrete class without changing Java code
    - Using annotations in the Java code partially violates that principle
  - faces-config.xml lets you specify Map elements
    - The annotation does not
- **Spring is best of all (?)**
  - For situations that warrant the extra complexity, Spring has nice JSF integration, so you can directly use Spring configuration files (e.g., applicationContext.xml). See separate lecture.

@ManagedProperty: Example

- **Idea**
  - Refactoring of banking example
  - Customer lookup service will be injected into main bean via @ManagedProperty
  - Lookup service will be application scoped and created at application load time
- **What you need**
  - Main bean

```
@ManagedProperty(value="#{lookupServiceBeanName}")
private CustomerLookupService service;
public void setService(…) { … }
```

  - Lookup service bean (the bean being injected)
    - @ManagedBean(eager=true)
      @ApplicationScoped

bank-lookup3.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</legend>
<h:form>
  Customer ID:
  <h:inputText value="#{bankingBean3.customerId}"/><br/>
  Password:
  <h:inputSecret value="#{bankingBean3.password}"/><br/>
  <h:commandButton value="Show Current Balance"
                   action="#{bankingBean3.showBalance}"/>
</h:form>
</fieldset>
…
</h:body></html>
```

Same as bank-lookup.xhtml except for bean name and some minor changes to text.

BankingBean3.java

```
@ManagedBean
public class BankingBean3 extends BankingBeanBase {
  @ManagedProperty(value="#{customerSimpleMap2}")
  private CustomerLookupService service;

  public void setService(CustomerLookupService service) {
    this.service = service;
  }
 public String showBalance() {
    if (!password.equals("secret")) {
      return("wrong-password3");
    }
    customer = service.findCustomer(customerId);
    if (customer == null) {
      …
  }
}
```

There is now no explicit reference to the concrete class that provides the lookup service. So, it is easier to change among test implementations and real implementations without changing this code. However, when you use annotations only, the bean name is often tied closely to the bean class. The bean name need not be the class name, but you still specify the name in the Java code. So, it is not clear that the annotation-based way of doing dependency injection is better than the JSF 1.*x* way of using faces-config.xml.

Even though you must put the @ManagedProperty before the field (instance variable), the setter is what is called by JSF. If the setter name does not match the field name, use @ManagedProperty(*name*…, value…).

CustomerSimpleMap2.java

```
@ManagedBean(eager=true)
@ApplicationScoped
public class CustomerSimpleMap2
       extends CustomerSimpleMap {
}
```

Since the lookup service is shared by all instances of the banking bean, it is declared application scoped. Since it might take some time to initialize the Map, it is instantiated at application load time via "eager". You are required to have @ApplicationScoped if you use eager=true.

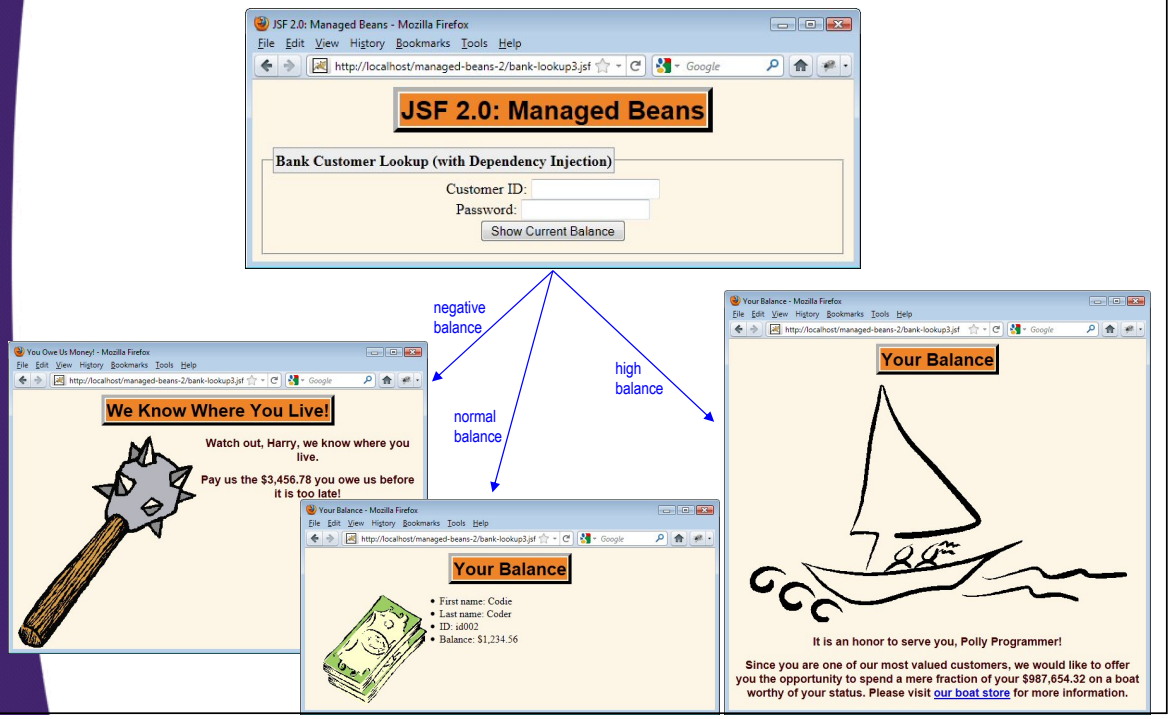
The only reason for the subclass (instead of just putting the annotations on CustomerSimpleMap) is so that I can use "eager" here but not in previous example that used CustomerSimpleMap. Also note that the lookup service is immutable (does not change), so we don't have to worry about synchronizing access to it.

normal-balance3.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: #{bankingBean3.customer.firstName}</li>
  <li>Last name: #{bankingBean3.customer.lastName}</li>
  <li>ID: #{bankingBean3.customer.id}</li>
  <li>Balance: $#{bankingBean3.customer.balanceNoSign}</li>
</ul>
…
</h:body></html>
```

Same as normal-balance.xhtml except for bean name.
negative-balance3.xhtml and high-balance3.xhtml are similar.

Results (Same Behavior as First Banking Example)



Wrap-Up

Summary

- **Custom bean names**
  - @ManagedBean(name="yourBeanName")
- **Scopes**
  - Session scope commonly used for user preferences and other user-specific data that must survive from request to request
    - @SessionScoped
- **The raw request and response objects**
  - Needed for general redirects, headers, long-lived cookies, etc.
    - Use static method calls with ExternalContext to get them
- **Dependency injection**
  - Code can be more flexible if services (and other classes that commonly change) are passed in instead of hard coded
    - We used @ManagedProperty, but also consider declaring in faces-config or using the Spring framework (not covered yet)

Questions?