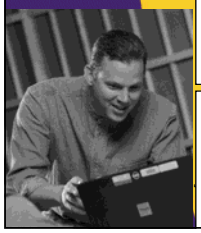
ui:repeat and Handling Variable-Length Data

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

- **Options for handling variable-length data**
  - Building strings or simple HTML from a bean property
  - Using a builtin component like h:dataTable
  - Making your own composite component
  - Looping with ui:repeat
- **Using ui:repeat**
  - Simple loops
  - Nested loops
  - varStatus
  - Conditional output
    - #{someCondition ? simpleVal1 : simpleVal2}
    - <h:outputText rendered="…" …/>
    - <ui:fragment rendered="…">…</ui:fragment>

Overview

Issue

- **Goal**
  - You want results pages to be simple and HTML-oriented
    - Separation of concerns
    - Allows Web page developers to build GUI
- **Problem**
  - What if the action controller method produces data whose length can change? How do you generate output without resorting to JSP scripting and explicit Java looping?
- **Solutions**
  - There are a number of alternatives, *all* of which would work well in some circumstances. The issue is how much control the Web page author needs.
    - We will cover ui:repeat in the most detail, but the other alternatives are also reasonable in real life

JSF Constructs for Handling Variable-Length Data

Simplest for Page Author

↓

Most Control for Page Author

- **Bean**
  - Have bean getter method spit out string or super-simple HTML based on a collection
- **h:dataTable**
  - Use a builtin component that builds a table from a collection. 3rd-party variations such as t:dataList (MyFaces/Tomahawk) give even more options.
- **Your own composite component**
  - Make own component that builds some HTML construct (e.g., <ul> list) from a collection
- **ui:repeat**
  - Do explicit looping in results page

Competing Concerns

- **Principles**
  - Simplicity is better in the .xhtml pages
  - Layout and formatting decisions should be made by the author of the results page, not by the Java programmer
- **General approach**
  - Use the simplest option that gives the Web page author enough control for the specific situation
- **Notes**
  - We only briefly survey composite components & h:dataTable here. We cover them in detail later in tutorial.
  - Although composite components are simpler than ui:repeat to *use* once they are created, they are more complex to *build* and often use ui:repeat internally. So, we cover them here *after* ui:repeat.

When to Use Which: Summary

- **Bean**
  - Write a getter method in the bean that turns collection into plain text or HTML.
    - Programmer knows #{programmer.languageList}.
    - Results in “Programmer knows Java, C++, and Ruby
- **When it works well**
  - You output plain text or very simple HTML.
  - You use the same output format several places, with no customization beyond what CSS provides.
- **When it works poorly**
  - The page author needs more control over the output format.

When to Use Which: Summary

- **h:dataTable**
  - Use the builtin component that turns a collection into an HTML table
    - <h:dataTable var="programmer" value="#{corp.hackers}">
    <h:column>#{programmer.firstName}</h:column>
    ….
</h:dataTable>
- **When it works well**
  - You want to build an HTML table out of the data
    - Where each entry in data corresponds to a table row
- **When it works poorly**
  - You want to build something other than an HTML table
  - Different parts of the table come from different sources
    - Although you could make new bean with composite data

When to Use Which: Summary

- **Your own composite component**
  - Make a new component that turns a collection into HTML (usually something other than a table)
    - <utils:list value="#{programmer.languages}"
    styleClass="some-css-name"/>
    - Result: <ul class="some-css-name">
Java
…
- **When it works well**
  - You want to build something other than a table
  - You can anticipate the options that page designer needs
- **When it works poorly**
  - Data is in a slightly different format than expected
  - Page author wants even a small change that was not anticipated by component author

When to Use Which: Summary

- **ui:repeat**
  - Use facelets looping to build the HTML inside page
    -
      <ui:repeat var="language" value="#{person.languages}">
      #{language}
      </ui:repeat>

- **When it works well**
  - The page designer needs explicit control over the result
  - One of the previous options is not sufficient
- **When it works poorly**
  - The page becomes so complex that it is hard to maintain by HTML-oriented page author

Example Notes

- **Data**
  - Normally, the data is produced in the action controller.
    - E.g., you collect a bank customer ID and month in a form, and then the button says
      <h:commandButton … action="#{user.findChanges}"/>
      where findChanges finds the deposits and withdrawals in the month and puts then into an array or List.
  - Here, we will hardcode the data for simplicity.
- **Order of topics**
  - Your own composite component is listed before ui:repeat because it is usually simpler for a page author to *use* a composite component than an explicit loop.
  - But, *building* a composite component is more complex and requires ui:repeat internally, so is covered after ui:repeat in this tutorial.

Using ui:repeat – Getting Started

Big Idea

- **Situation**
  - You have variable-length data, but don't want to output an HTML table.
  - You can't easily build a composite component that gives the page designer enough flexibility.
- **Approach**
  - Use ui:repeat almost exactly as you would use JSTL's c:forEach.
- **Pros**
  - Gives page author explicit control
  - Far simpler and more readable than a JSP scripting loop
- **Cons**
  - Regular HTML has no loops, so page is complex

Syntax Summary

- **Basics**

```
<ui:repeat var="someVar"
           value="#{someBean.someCollection}">
    <someHTML>#{someVar.someProperty}</someHTML>
</ui:repeat>
```

- **Analogous Java code**

```
for(SomeType someVar: someCollection) {
  doSomethingWith(someVar);
}
```

- **Warnings**
  - Value can be only array or List (or ResultSet). Not Map. Cannot use int[] or double[] or other array of primitives
    - Use Integer[] or Double[] instead
  - JSF 2.3 promises to support Map and Iterable

Steps to Using ui:repeat

- **Include the facelets namespace**
  - <html … xmlns:ui="http://xmlns.jcp.org/jsf/facelets">
- **Make collection accessible**
  - Make getter method that returns List, array, or ResultSet
    - E.g., getColors
- **Use loop in facelets page**

```
<ul>
<ui:repeat var="color" value="#{test.colors}">
   <li>#{color}</li>
</ui:repeat>
</ul>
```

Why Not JSTL?

- **Question**
  - We know JSTL and c:forEach, why learn something new?
- **Answers**
  - c:forEach runs when the component tree is being built. ui:repeat runs when the tree is being rendered. The latter is when you usually want it to run.
  - ui:repeat is virtually identical in syntax and behavior to c:forEach anyhow, so if you know c:forEach, it is *very* simple to learn ui:repeat
    - For the data, just use "value" instead of "items"
- **Caveat**
  - You need c:forEach when you want ui:include inside a loop, since ui:include runs when tree is being built
    - For info on ui:include, see tutorial section "Page Templating with Facelets".

Simplifying Testing of ui:repeat Features

- **Usual usage**
  - Action controller method gets list or array back from business logic, results page needs to display it
- **Usage when practicing**
  - Make managed bean with getter method that returns list or array. Test it in one standalone JSF page.
- **Example**

```
<ui:repeat var="something"
           value="#{myBean.myProperty}">
  … #{something} …
</ui:repeat>
```

  - If JSF cannot find myBean in existing scope, it instantiates it on the spot (assuming it is a managed bean).

Simplifying Testing of ui:repeat Example

Bean	Standalone Test Page

`@ManagedBean`
`public class Test {`
`  private static String[] colors =`
`  { "red", "green", "blue" };`
 	
`  public String[] getColors() {`
`  return(colors);`
`  }`
`}`
	`<!DOCTYPE ...>`
	`<html xmlns="http://www.w3.org/1999/xhtml"`
	`  xmlns:h="http://xmlns.jcp.org/jsf/html"`
	`  xmlns:ui="http://xmlns.jcp.org/jsf/facelets">`
	`...`
	`<ul>`
	`  <ui:repeat var="color" value="#{test.colors}">`
	`  <li>#{color}</li>`
	`  </ui:repeat>`
	`</ul>`
	`...`

- red
- green
- blue

Using ui:repeat – Basics

Simple Loop: Facelets Code (Top)

```
<!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"
      xmlns:ui="http://xmlns.jcp.org/jsf/facelets">
<h:head><title>Looping with ui:repeat</title>
<link href="./css/styles.css"
      rel="stylesheet" type="text/css"/>
<style type="text/css">
  .evenLang { color: blue }
  .oddLang { color: red }
</style>
</h:head>
<h:body>
```

We use ui:repeat and possibly other ui: elements, so we have to add this namespace

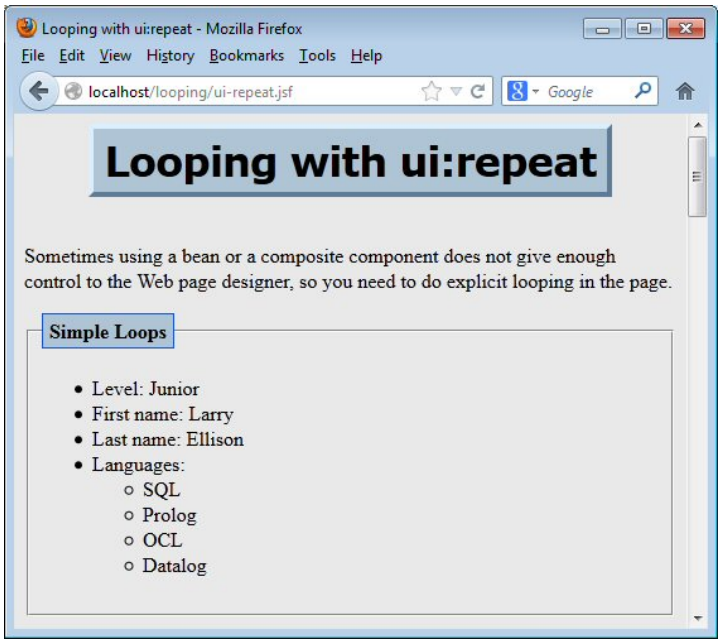
These styles are used in a later example (re the varStatus attribute)

Simple Loop: Facelets Code (ui:repeat Part)

```
<ul>
  <li>Level: #{person1.level}</li>
  <li>First name: #{person1.firstName}</li>
  <li>Last name: #{person1.lastName}</li>
  <li>Languages:
    <ul>
    <ui:repeat var="language" value="#{person1.languages}">
      <li>#{language}</li>
    </ui:repeat>
    </ul>
  </li>
</ul>
```

Code for the person1 managed bean shown earlier. The getLanguages method returns String[].

Simple Loop: Results



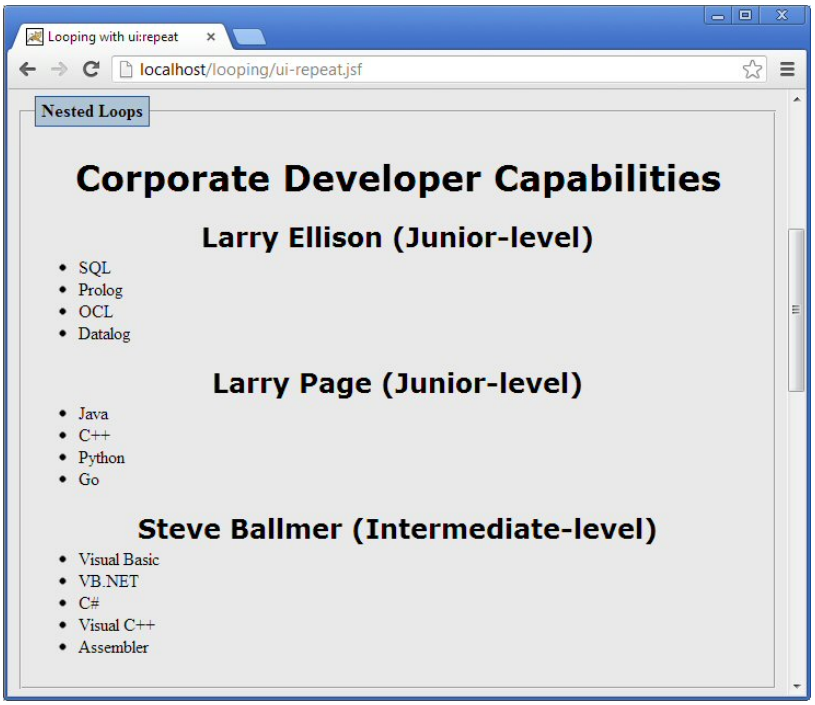
Nested Loop: Java Code

```
…

@ManagedBean(eager=true)
public class Company1 {
  private List<Programmer> programmers;

  public Company1() {
    programmers = new ArrayList<>();
    programmers.add(new Person1());
    programmers.add(new Person2());
    programmers.add(new Person3());
  }

  public List<Programmer> getProgrammers() {
    return(programmers);
  }
}
```

Nested Loop: Facelets Code

```
<ui:repeat var="programmer" value="#{company1.programmers}">
<h2>#{programmer.firstName} #{programmer.lastName}
    (#{programmer.level}-level)
</h2>
<ul style="margin-top: -1em;">
<ui:repeat var="language" value="#{programmer.languages}">
  <li>#{language}</li>
</ui:repeat>
</ul>
</ui:repeat>
```


Nested Loop: Results



Conditional Text Inside Loops

Conditional Text in JSF

- **Alternatives**
  - #{someCondition ? simpleVal1 : simpleVal2}
  - <h:outputText value="#{someValue}"
    rendered="#{someCondition}"/>
    - Or, in general, use h:blah and the “rendered” attribute
  - <ui:fragment rendered="#{someCondition}">
    <someHTML>…</someHTML>
    </ui:fragment>
- **Note to JSTL developers**
  - c:if and c:choose don’t interact properly with ui:repeat because they run when component tree is built, not when it is rendered.
    - So, don’t use the JSTL conditional evaluation tags

Conditional Text with #{ condition ? val1 : val2 }

- **Idea**
  - The EL directly supports limited conditional output via the ternary operator (test ? thenResult : elseResult). Supply a boolean for the test, put conditional content after the "?" and/or the ":". Values can be literal strings or EL expressions, but they cannot contain HTML tags.
- **Examples**
  - <td class="#{customer.balance < 0 ? 'red': 'black'}">
  - #{ !status.last ? ',' : '' }
- **When used**
  - When you are outputting simple text (no HTML).

If you want to output HTML, you could use the ternary operator within h:outputText and supply escape="false". But in that case, one of the other two upcoming alternatives is probably simpler.

Conditional Text with h:outputText and "rendered"

- **Idea**
  - Pass a boolean to the "rendered" attribute, put conditional content in "value" attribute. The value can be a literal string or an EL expression, but the literal string cannot contain HTML tags.
- **Examples**
  - <h:outputText rendered="#{!status.last}" value=","/>
  - <h:outputText rendered="#{status.index > 5}"
    value="#{user.someWarning}"
    escape="false"/>

The assumption here is that the getSomeWarning method outputs a string containing HTML tags. If so, the escape="false" is needed to prevent JSF from turning the < into < and so forth.

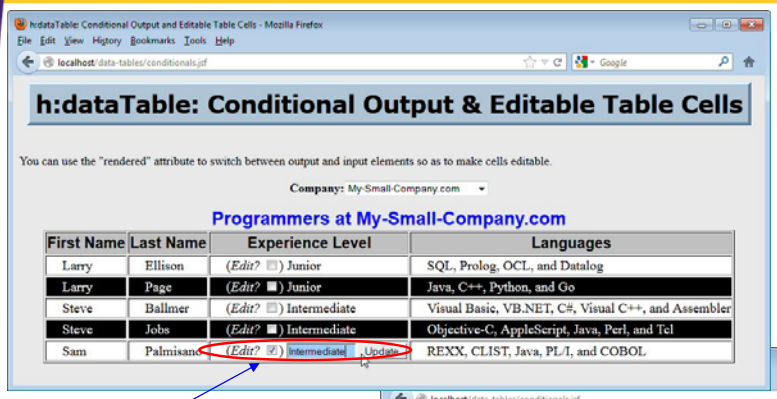
- **When used**
  - When you are outputting simple text (no HTML) or when the HTML comes from a bean.

More on “rendered” Attribute

- **Almost all h:*blah* elements use “rendered”**
  - So, you can insert almost any JSF element conditionally.
- **Example**
  - Insert either textfield followed by button *or* simple value (full example in tutorial section on h:dataTable)

```
<h:inputText value="#{programmer.level}" size="12"
                    rendered="#{programmer.levelEditable}"/>
<h:commandButton value="Update"
                          rendered="#{programmer.levelEditable}">
      <f:ajax render="@form" execute="@form"/>
</h:commandButton>
<h:outputText value="#{programmer.level}"
                        rendered="#{!programmer.levelEditable}"/>
```


Example: Use of “rendered”

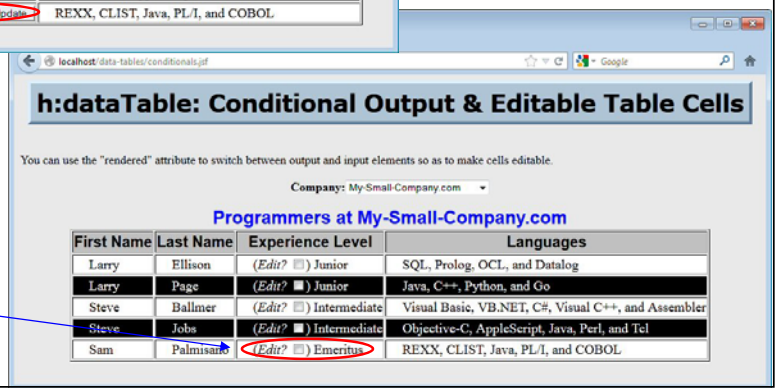


Idea

If checkbox selected, use textfield and “Update” button. Otherwise just show current value.

Full example in tutorial section on h:dataTable

After clicking on checkbox



After typing in “Emeritus” and clicking “Update”

Conditional Text with ui:fragment

- **Idea**
  - Pass a boolean to the "rendered" attribute, put conditional content in body content. The value can be a literal string or an EL expression, and the literal string *can* contain HTML tags.
- **Example**
  - <ui:fragment rendered="#{!status.last}">
    ,
    </ui:fragment>

    Outputs a bold comma after every entry except the last
- **When used**
  - When you are outputting literal HTML.
    - Can always be used in lieu of h:outputText, but if no HTML, h:outputText is more succinct.

Using ui:repeat – Advanced Attributes

Feature Summary

- **Additional attributes**

```
<ui:repeat var="someVar"
           value="#{someBean.someCollection}"
           varStatus="statusVariable"
           offset="…"
           size="…"
           step="…">
  …
</ui:repeat>
```


ui:repeat Attributes

Attribute Name	Description	c:forEach Equivalent
var	String giving the local variable name that will refer to each element in the collection. E.g.: <ui:repeat var="name" value="#{party.names}">	var
value	EL expression specifying the collection itself.	items
varStatus	String giving a variable name that will refer to a status object. The status object has the following properties: • begin, end, step: values of offset, size, and step attributes • index: the current index (int) • first/last: is this the first/last iteration? (boolean) • even/odd: is this even/odd iteration? (boolean) (1st iteration is 0: even) <ui:repeat … varStatus="status"> Do something with #{status.first} or #{status.even}, etc. </ui:repeat>	varStatus
offset	An int specifying how far into the collection to start. Default is 0. E.g., offset="1" means to skip the first (0th) element.	begin
size	An int specifying how far down the collection to go. Default is the end of the collection.	end
step	An int specifying how far to jump down the collection after each item. Default is 1.	step

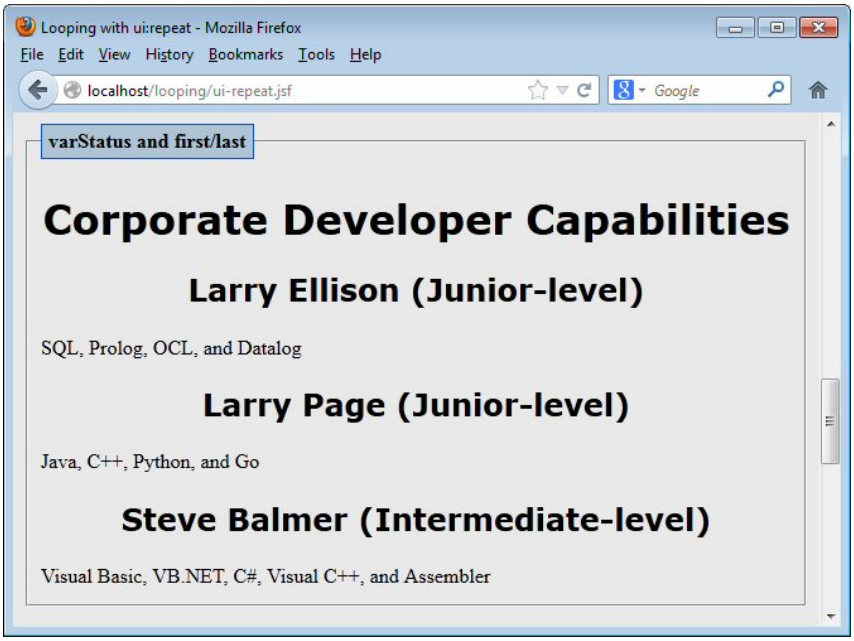
Example 1: Overview

- **Goal**
  - From a list of strings, generate "String1, String2, …, and String*N*".
- **Approach**
  - Track iteration status
    - <ui:repeat … varStatus="status">
  - Output a comma *except* in last iteration
    - <h:outputText rendered="#{!status.last}" value=","/>
  - Output "and" *only* in last iteration
    - <h:outputText rendered="#{status.last}" value=" and "/>

Example 1: Facelets Code

```
<ui:repeat var="programmer" value="#{company1.programmers}">
<h2>#{programmer.firstName} #{programmer.lastName}
    (#{programmer.level}-level)
</h2>
  <ui:repeat var="language" value="#{programmer.languages}"
             varStatus="status">
      <h:outputText value=" and "
                    rendered="#{status.last}"/>
    #{language}<h:outputText value=","
                             rendered="#{!status.last}"/>
  </ui:repeat>
</ui:repeat>
```

Example 1: Results



Quick Aside: StringJoiner in Java 8

- **Big ideas**
  - Java 8 added new StringJoiner class that builds delimiter-separated Strings, with optional prefix and suffix
  - Java 8 also added static "join" method to the String class; it uses StringJoiner internally
- **Quick examples (result: "Java, Lisp, Ruby")**
  - Explicit StringJoiner with no prefix or suffix

```
StringJoiner joiner1 = new StringJoiner(", ");
String result1 =
    joiner1.add("Java").add("Lisp").add("Ruby").toString();
```

  - Usually easier: String.join convenience method

```
String result2 = String.join(", ", "Java", "Lisp", "Ruby");
String[] languages = {"Java", "Lisp", "Ruby"};
String result3 = String.join(", ", languages);
```

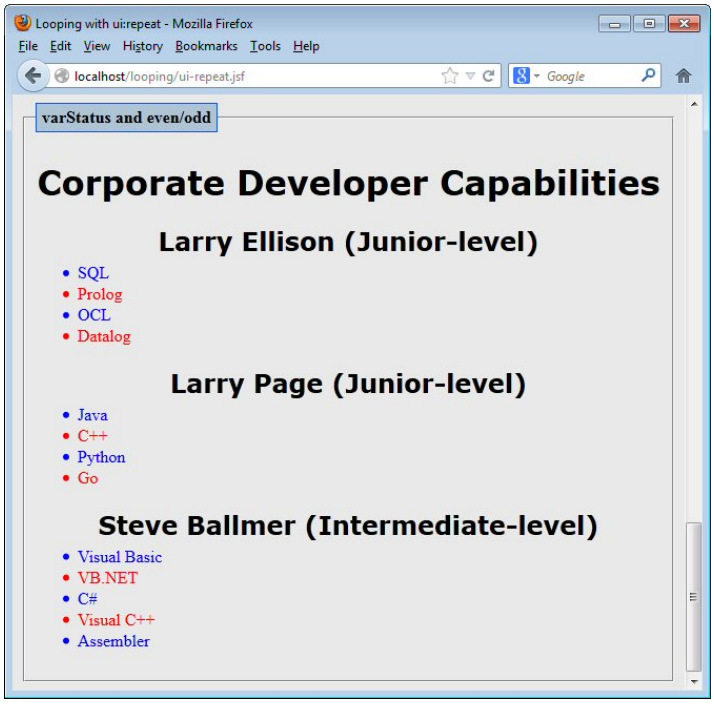
Example 2: Overview

- **Goal**
  - From a list of strings, generate a bulleted (<ul>) list.
    - Have every other entry in a different style
- **Approach**
  - Track iteration status
    - <ui:repeat … varStatus="status">
  - Output one type of li in even iterations
    - <ui:fragment rendered="#{status.even}">
      <li class="evenLang">#{language}
      </ui:fragment>
  - Output another type of li in odd iterations
    - <ui:fragment rendered="#{status.odd}">
      <li class="oddLang">#{language}
      </ui:fragment>
  - Use style sheet to map the styles
    - evenLang and oddLang

Example 2: Facelets Code

```
<ui:repeat var="programmer" value="#{company1.programmers}">
<h2>#{programmer.firstName} #{programmer.lastName}
    (#{programmer.level}-level)
</h2>
<ul style="margin-top: -1em;">
<ui:repeat var="language" value="#{programmer.languages}"
           varStatus="status">
  <ui:fragment rendered="#{status.even}">
    <li class="evenLang">#{language}</li>
  </ui:fragment>
  <ui:fragment rendered="#{!status.even}">
    <li class="oddLang">#{language}</li>
  </ui:fragment>
</ui:repeat>
</ul>
</ui:repeat>
```

Example 2: Results



Example 2: Alternative

- **Example did this**

```
<ui:fragment rendered="#{status.even}">
   <li class="evenLang">#{language}</li>
</ui:fragment>
<ui:fragment rendered="#{!status.even}">
   <li class="oddLang">#{language}</li>
</ui:fragment>
```

- **It could have done this**

```
<li class="#{status.even ? 'evenLang' : 'oddLang'}">
   #{language}
</li>
```

- **General point**
  - If you want to conditionally include large chunks of literal HTML, use ui:fragment, not h:outputText.

Wrap-Up

Summary

- **ui:repeat basics**

```
<ul>
<ui:repeat var="color" value="#{item.availableColors}">
   <li>#{color}</li>
</ui:repeat>
</ul>
```

- **Other ui:repeat capabilities**
  - varStatus attribute
    - Especially first, last, even, and odd boolean properties
  - Conditional output
    - Use “rendered” attribute of h:outputText or ui:fragment
- **Consider alternatives to ui:repeat**
  - Bean getter method that builds result (only if *very* simple)
  - h:dataTable – covered in separate tutorial section
  - Composite component – covered in separate tutorial section
    - Especially one that uses ui:repeat internally

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