A Brain-Friendly Guide

2nd Edition
New Mock Exam Included

Head First Servlets & JSP™

Passing the Sun Certified Web Component Developer Exam

Test yourself with more than 200 realistic exam questions

Updated to cover the latest version of the SCWCD exam for J2EE 1.5

Watch it!

Avoid deadly traps & gotchas on the 1.5 exam

Use c:out to get your message to the world

Learn how Ted improved his appeal with dynamic attributes

Fool around in the Custom Tag Library

O'REILLY®

Bryan Basham, Kathy Sierra & Bert Bates

Head First Servlets and JSP™

Second Edition

by Bryan Basham, Kathy Sierra, and Bert Bates

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly Media books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*safari.oreilly.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Series Creators:	Kathy Sierra, Bert Bates
Series Editor:	Brett D. McLaughlin
Design Editor:	Louise Barr
Cover Designers:	Edie Freedman, Steve Fehler, Louise Barr
Production Editor:	Sanders Kleinfeld
Indexer:	Julie Hawks
Interior Decorators:	Kathy Sierra and Bert Bates
Servlet Wrangler:	Bryan Basham
Assistant to the Front Controller:	Bert Bates

Printing History:

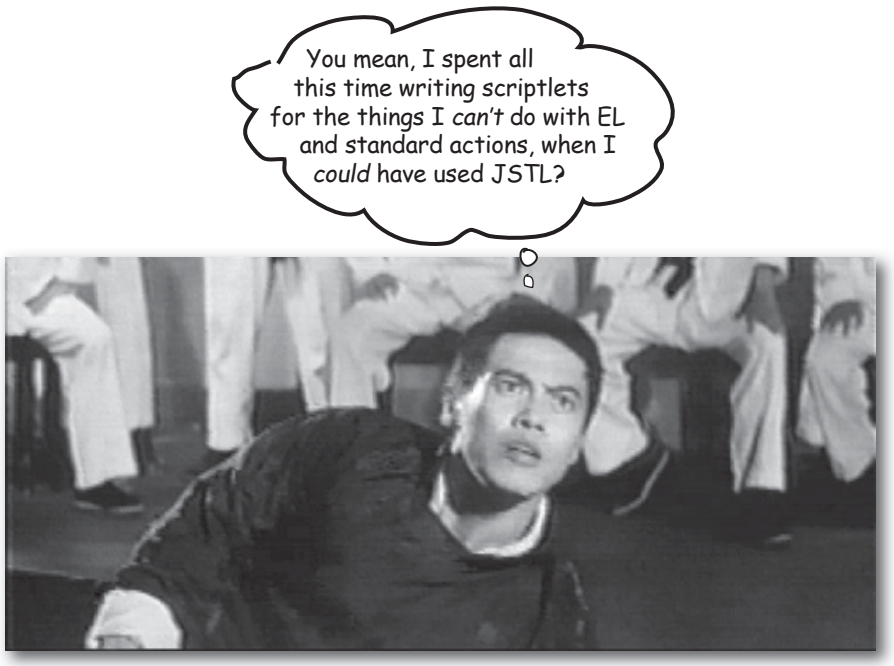
August 2004: First Edition.

March 2008: Second Edition.

ISBN: 978-0-596-51668-0

[M]

Custom tags are powerful



Sometimes you need more than EL or standard actions.

What if you want to loop through the data in an array, and display one item per row in an HTML table? You *know* you could write that in two seconds using a for loop in a scriptlet. But you're trying to get away from scripting. No problem. When EL and standard actions aren't enough, you can use *custom tags*. They're as easy to use in a JSP as standard actions. Even better, someone's already written a pile of the ones you're most likely to need, and bundled them into the JSP Standard Tag Library (JSTL). In *this* chapter we'll learn to *use* custom tags, and in the next chapter we'll learn to create our own.

Building JSP pages using tag libraries

9.1 Describe the syntax and semantics of the 'taglib' directive: for a standard tag library, for a library of Tag Files.

9.2 Given a design goal, create the custom tag structure to support that goal.

9.3 Identify the tag syntax and describe the action semantics of the following JSP Standard Tag Library (JSTL v1.1) tags: (a) core tags: out, set, remove, and catch, (b) conditional tags: if, choose, when, and otherwise, (c) iteration tags: forEach, and (d) URL-related: url.

Coverage Notes:

All of the objectives in this section are covered in this chapter, although some of the content is covered again in the next chapter (Developing Custom Tags).

Installing the JSTL 1.1

The JSTL 1.1 is NOT part of the JSP 2.0 specification! Having access to the Servlet and JSP APIs doesn't mean you have access to JSTL.

Before you can use JSTL, you need to put two files, "jstl.jar" and "standard.jar" into the WEB-INF/lib directory of your web app. That means each web app needs a copy.

In Tomcat 5, the two files are already in the example applications that ship out-of-the-box with Tomcat, so all you need to do is copy them from one directory and put them into your own app's WEB-INF/lib directory.

Copy the files from the Tomcat examples at:

webapps/jsp-examples/WEB-INF/lib/jstl.jar
webapps/jsp-examples/WEB-INF/lib/standard.jar

And place it in your own web app's WEB-INF/lib directory.

EL and standard actions are limited

What happens when you bump into a brick wall? You can go back to scripting, of course—but you know that's not the path.

Developers usually want *way* more standard actions or—even better—the ability to create their *own* actions.

That's what ***custom tags*** are for. Instead of saying <jsp:setProperty>, you want to do something like <my:doCustomThing>. And you can.

But it's not that easy to create the support code that goes behind the tag. For the JSP page creator, custom tags are much easier to use than scripting. For the Java programmer, however, building the custom tag *handler* (the Java code invoked when a JSP uses the tag) is tougher.

Fortunately, there's a standard library of custom tags known as the **JSP Standard Tag Library** (JSTL 1.1). Given that your JSP shouldn't be doing a bunch of business logic anyway, you might find that the JSTL (combined with EL) is all you'll ever need. Still, there could be times when you need something from, say, a custom tag library developed specifically for your company.

In *this* chapter, you'll learn how to use the core JSTL tags, as well as custom tags from other libraries. In the *next* chapter, we'll learn how to actually build the classes that handle calls to the custom tags, so that you can develop your own.

The case of the disappearing HTML (reprised)

On page 384, you saw how EL sends the raw string of content directly to the response stream:

```
<div class='tipBox'>
  <b>Tip of the Day:</b> <br/> <br/>
  ${pageContent.currentTip}
</div>
```

Remember this? The tags didn't show up as text, but got rendered as an empty space that was bolded.

What we got

```
<div class='tipBox'>
  <b>Tip of the Day:</b> <br/>
  <b></b> tags make things bold!
</div>
```

This comes out as an "invisible" bolded empty space.

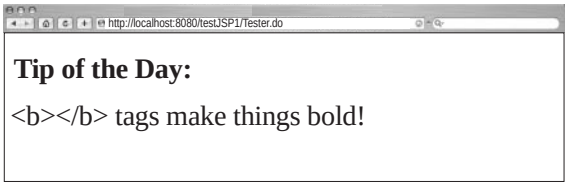
Rendered as

What we want

```
<div class='tipBox'>
  <b>Tip of the Day:</b> <br/> <br/>
  &lt;b&gt;&lt;/b&gt; tags make things bold!
</div>
```

< is rendered as "<", and > is rendered as ">".

Rendered as



What we need is a way to convert those angle brackets into something the browser will render as angle brackets, and there are two ways to do this. Both use a static Java method that converts HTML special characters into their entity format:

Use an EL function

```
<div class='tipBox'>
  <b>Tip of the Day:</b> <br/> <br/>
  ${fn:convEntity(pageContent.currentTip)}
</div>
```

Use a Java helper method

```
<div class='tipBox'>
  <b>Tip of the Day:</b> <br/> <br/>
  ${pageContent.convertedCurrentTip}
</div>
```

Here's the helper method to make this one work.

```
public String getConvertedCurrentTip() {
  return HTML.convEntity(getCurrentTip());
}
```

There's a better way: use the <c:out> tag

Whichever approach you use, it's a bit unclear exactly what's going on... and you may have to write that helper method for all your servlets. Luckily, there's a better way. The <c:out> tag is perfect for the job. Here's how conversion works:

You can explicitly declare the conversion of XML entities

If you know or think you might run into some XML entities that need to be displayed, and not just rendered, you can use the escapeXml attribute on c:out. Setting this to true means that any XML will be converted to something the web browser will render, angle brackets and all:

```
<div class='tipBox'>
  <b>Tip of the Day:</b> <br/> <br/>
  <c:out value='${pageContent.currentTip}' escapeXml='true' />
</div>
```

Your HTML is treated as XHTML, which in turn is XML... so this affects HTML characters, too.

You can explicitly declare NO conversion of XML entities

Sometimes, you want just the opposite behavior. Maybe you're building a page that takes content, and you want to display that content with HTML formatting. In that case, you can turn off XML conversion:

```
<div class='tipBox'>
  <b>Tip of the Day:</b> <br/> <br/>
  <c:out value='${pageContent.rawHTML}' escapeXml='false' />
</div>
```

This is equivalent to what we had before... any HTML tags are evaluated, not displayed as text.

Conversion happens by default

The **escapeXml** attribute defaults to true, so you can leave it out if you want. A **c:out** tag without an **escapeXML** attribute is just the same as a **c:out** tag with **escapeXML** set to "true."

```
<div class='tipBox'>
  <b>Tip of the Day:</b> <br/> <br/>
  <c:out value='${pageContent.currentTip}' />
</div>
```

This is actually identical in functionality to this.

there are no Dumb Questions

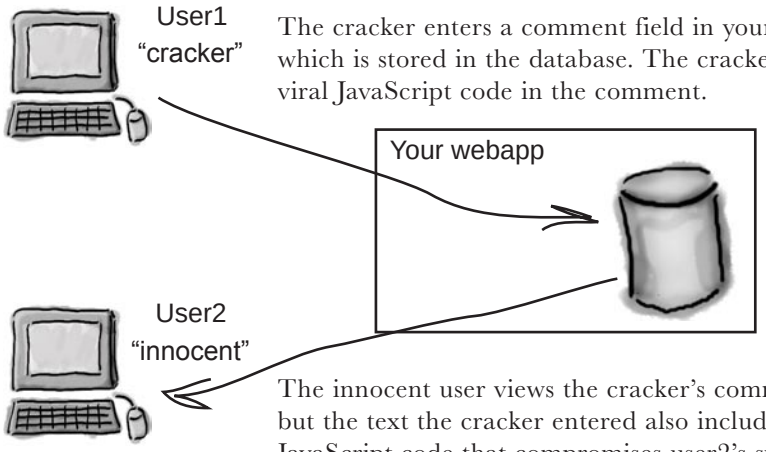
Q: **Which HTML special characters are converted?**

A: It turns out this conversion is rather simple. There are only five characters that require escaping: **<, >, &,** and the two quote symbols, single and double **".** All of these are converted into the equivalent HTML entities. For example, **<** becomes **`<`**, **&** becomes **`&`,** and so on.

Character	Character Entity Code
<	`<`
>	`>`
&	`&`
'	`&#039;`
"	`&#034;`

Q: **Last month my company hired a web consultant to audit our web application. She noticed that we were using EL everywhere to output strings entered by users. She said this was a security risk and recommended we output all user strings using the c:out tag. What gives?**

A: Your consultant was right. The security risk she is referring to is called **cross-site hacking** or **cross-site scripting**. The attack is sent from one user to another user's web browser using your webapp as the delivery mechanism.



Using the c:out tag to render the text of users prevents cross-site hacking of this form by displaying the <script> tags and the JS code in user2's web browser. This prevents the JS code from being interpreted by the browser, foils the attack from user1.

Q: **What happens if value of the EL expression is null?**

A: Good question. You know an EL expression **`${evalsToNull}`** generates an empty string in the response output, and so will **`<c:out value="${evalsToNull}"/>`**.

But that's not the end of the story with **`c:out`**. The **`c:out`** tag is smart, and it recognizes when the value is null and can perform a special action. That action is to provide a default value...

Null values are rendered as blank text

Suppose you have a page that welcomes the user by saying "Hello <user>." But lately, users haven't been logging in, and the output looks pretty odd:

EL prints nothing if user is null

```
<b>Hello ${user}.</b>
```

Renders as

```
<b>Hello .</b>
```

A JSP expression tag prints nothing if user is null

```
<b>Hello <%= user %>.</b>
```

Renders as

```
<b>Hello .</b>
```

Since ${user} and <%= user %> evaluate to null, you get an empty space between "Hello" and the "." Pretty strange looking...

Set a default value with the default attribute

Suppose you want to show these anonymous users a message that says, "Hello guest." This is a perfect place to use a default value with the **`c:out`** tag. Just add a **`default`** attribute, and provide the value you want to print if your expression evaluates to null:

<c:out> provides a default attribute

This value is output if the value attribute evaluates to null.

```
<b>Hello <c:out value='${user}' default='guest' />.</b>
```

Renders as

```
<b>Hello guest.</b>
```

Now the default value is inserted... perfect.

Or you can do it this way:

```
<b>Hello <c:out value='${user}'>guest</c:out></b>
```

Looping without scripting

Imagine you want something that loops over a collection (say, an array of catalog items), pulls out one element at a time, and prints that element in a dynamically-generated table row. You can't possibly hard-code the complete table—you have no idea how many rows there will be at runtime, and of course you don't know the values in the collection. The <c:forEach> tag is the answer. This does require a very slight knowledge of HTML tables, but we've included notes here for those who aren't familiar with the topic.

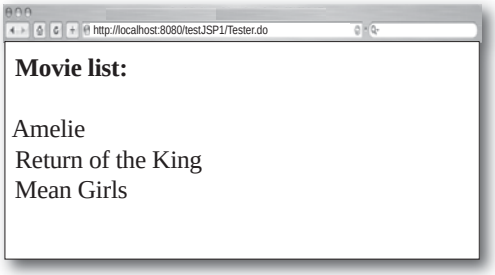
By the way, on the exam you *are* expected to know how to use <c:forEach> with tables.

Servlet code

```
...
String[] movieList = {"Amelie", "Return of the King", "Mean Girls"};
request.setAttribute("movieList", movieList);
...
```

Make a String[] of movie names, and set the array as a request attribute.

What you want



In a JSP, *with* scripting

```
<table>
<% String[] items = (String[]) request.getAttribute("movieList");
   String var=null;
   for (int i = 0; i < items.length; i++) {
      var = items[i];
 %>
  <tr><td><%= var %></td></tr>
  <% } %>
</table>
```

<c:forEach>

The <c:forEach> tag from the JSTL is perfect for this—it gives you a simple way to iterate over arrays and collections.

(We'll talk about this taglib directive later in the chapter.)

JSP code

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<html><body>
<strong> Movie list:</strong>
<br><br>

<table>
  <c:forEach var="movie" items="${movieList}" >
    <tr>
      <td>${movie}</td>
    </tr>
  </c:forEach>
</table>

</body></html>
```

Loops through the entire array (the "movieList" attribute) and prints each element in a new row. (This table has just one column per row.)

Crash refresher on HTML tables

<tr> stands for Table Row.

<td> stands for Table Data.

```
<table>
  <tr> <td>data for this cell</td> <td>data for this cell</td> <td>data for this cell</td> </tr>
  <tr> <td>data for this cell</td> <td>data for this cell</td> <td>data for this cell</td> </tr>
  <tr> <td>data for this cell</td> <td>data for this cell</td> <td>data for this cell</td> </tr>
</table>
```

Tables are pretty straightforward. They've got *cells*, arranged into *rows* and *columns*, and the data goes inside the cells. The trick is telling the table how many rows and columns you want.

Rows are defined with the <tr> (Table Row) tag, and columns are defined with the <td> (Table Data) tag. The number of rows comes from the number of <tr> tags, and the number of columns comes from the number of <td> tags you put inside the <tr></tr> tags.

Data to print/display goes only inside the <td> </td> tags!

Deconstructing <c:forEach>

The <c:forEach> tag maps nicely into a for loop—the tag repeats the *body* of the tag *for each* element in the collection (and we use "collection" here to mean either an array or Collection or Map or comma-delimited String).

The key feature is that the tag assigns each element in the collection to the variable you declare with the *var* attribute.

The <c:forEach> tag

The variable that holds each ELEMENT in the collection. Its value changes with each iteration.

```
<c:forEach var="movie" items="${movieList}" >

    ${movie}

</c:forEach>
```

The actual thing to loop over (array, Collection, Map, or a comma-delimited String).

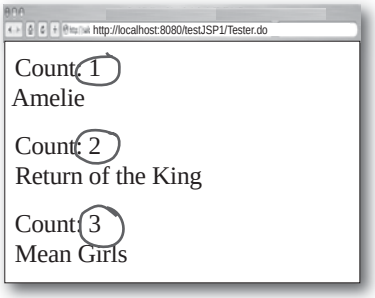
```
String[] items = (String[]) request.getAttribute("movieList");
for (int i = 0; i < items.length; i++) {
   String movie = items[i];
   out.println(movie);
}
```

Getting a loop counter with the optional varStatus attribute

varStatus makes a new variable that holds an instance of javax.servlet.jsp.jstl.core.LoopTagStatus.

```
<table>
  <c:forEach var="movie" items="${movieList}" varStatus="movieLoopCount" >
     <tr>
         <td>Count: ${movieLoopCount.count}</td>
     </tr>
     <tr>
         <td>${movie} <br><br></td>
     </tr>
  </c:forEach>
</table>
```

Helpfully, the LoopTagStatus class has a count property that gives you the current value of the iteration counter. (Like the "i" in a for loop.)



You can even nest <c:forEach> tags

What if you have something like a collection of collections? An array of arrays? You can nest <c:forEach> tags for more complex table structures. In this example, we put String arrays into an ArrayList, then make the ArrayList a request attribute. The JSP has to loop through the ArrayList to get each String array, then loop through each String array to print the actual elements of the array.

Servlet code

```
String[] movies1 = {"Matrix Revolutions", "Kill Bill", "Boondock Saints"};
String[] movies2 = {"Amelie", "Return of the King", "Mean Girls"};
java.util.List movieList = new java.util.ArrayList();
movieList.add(movies1);
movieList.add(movies2);
request.setAttribute("movies", movieList);
```

JSP code

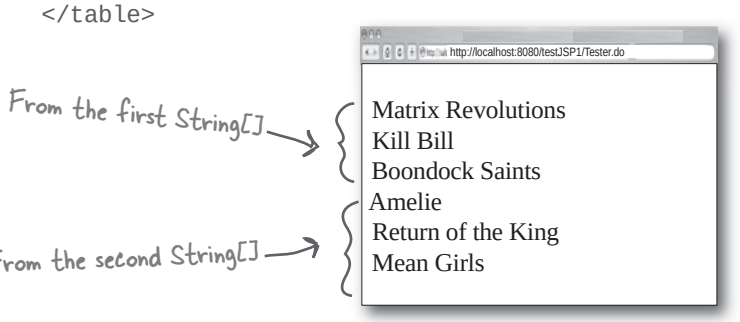
```
<table>

<c:forEach var="listElement" items="${movies}" >

   <c:forEach var="movie" items="${listElement}" >
     <tr>
       <td>${movie}</td>
     </tr>
   </c:forEach>

</c:forEach>

</table>
```

The ArrayList request attribute

outer loop

inner loop

One of the String arrays that was assigned to the outer loop's "var" attribute.



there are no Dumb Questions

Q: How did you know that the "varStatus" attribute was an instance of whatever that was, and how did you know that it has a "count" property?

A: Ahhhh... we looked it up.

It's all there in the JSTL 1.1 spec. If you don't have the spec already, go download it NOW (the intro of this book tells you where to get the specs covered on the exam). It is THE reference for all the tags in the JSTL, and tells you all the possible attributes, whether they're optional or required, the attribute type, and any other details on how you use the tag.

Everything you need to know about these tags (for the exam) is in this chapter. But some of the tags have a few more options than we cover here, so you might want to have a look in the spec.

Q: Since you know more than you're telling about this tag... does it give you a way to change the iteration steps? In a real Java for loop, I don't have to do i++, I can do i +=3, for example, to get every third element instead of every element...

A: Not a problem. The <c:forEach> tag has optional attributes for *begin*, *end* (in case you want to iterate over a subset of the collection), and *step* if you want to skip over some elements.

Q: Is the "c" in <c:forEach> a required prefix?

A: Well, *some* prefix is required, of course; all tags and EL functions must have a prefix to give the Container the namespace for that tag or function name. But you don't HAVE to name the prefix "c". It's just the standard convention for the set of tags in JSTL known as "**c**ore". We recommend using something *other* than "c" as a prefix, whenever you want to totally confuse the people you work with.

Watch it!

The "var" variable is scoped to ONLY the tag!

That's right, tag scope. No this isn't a full-fledged scope to which you can bind attributes like the other four—page, request, session, and application. Tag scope simply means that the variable was declared INSIDE a loop.

And you already know what that means in Java terms. You'll see that for most other tags, a variable set with a "var" attribute will be visible to whatever scope you specifically set (using an optional "scope" attribute), OR, the variable will default to page scope.

So don't be fooled by code that tries to use the variable somewhere BELOW the end of the <c:forEach> body tag!

```
<c:forEach var="foo" items="${fooList}" >
     ${foo}   <-- OK
</c:forEach>

${foo}   <-- NO!! The "foo" variable is out of scope!
```

It might help to think of tag scope as being just like block scope in plain old Java code. An example is the for loop you all know and love:

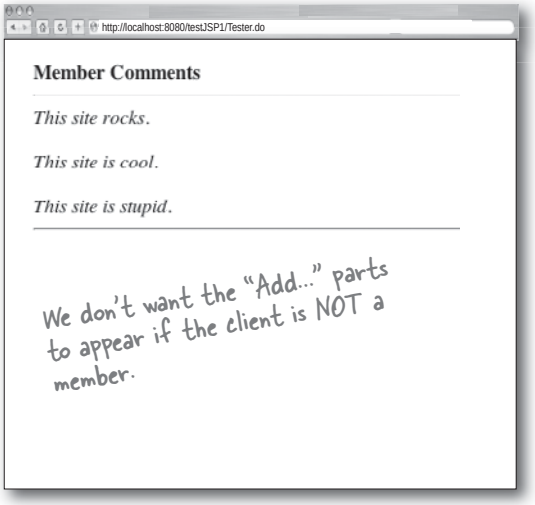
```
for (int i = 0; i < items.length; i++) {
  x + i;
}
doSomething(i);   // NO!! The "i" variable is out of scope!
```

Doing a conditional include with <c:if>

Imagine you have a page where users can view comments from other users. And imagine that members can also post comments, but non-member guests cannot. **You want *everyone* to get the same page, but you want *members* to "see" more things on the page.** You want a conditional <jsp:include > and of course, you don't want to do it with scripting!

What members see:

What NON-members see:



JSP code

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<html><body>
<strong>Member Comments</strong> <br>
<hr>${commentList}<hr>

<c:if test="${userType eq 'member' }" >
   <jsp:include page="inputComments.jsp"/>
</c:if>
</body></html>
```

Assume a servlet somewhere set the userType attribute, based on the user's login information.

Yes, those are SINGLE quotes around 'member'. Don't forget that you can use EITHER double or single quotes in your tags and EL.

Included page ("inputComments.jsp")

```
<form action="commentsProcess.jsp" method="post">
Add your comment: <br>
<textarea name="input" cols="40" rows="10"></textarea> <br>
<input name="commentSubmit" type="button" value="Add Comment">
</form>
```

But what if you need an *else*?

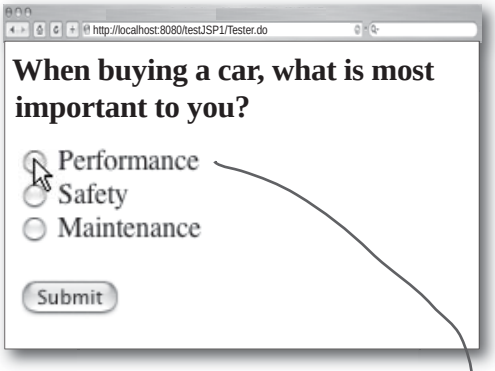
What if you want to do *one* thing if the condition is true, and a *different* thing if the condition is false? In other words, what if we want to show either one thing *or* the other, but *nobody* will see both? The <c:if> on the previous page worked fine because the logic was: *everybody* sees the first part, and then if the test condition is true, show a little extra.

But now imagine this scenario: you have a car sales web site, and ***you want to customize the headline that shows up on each page, based on a user attribute*** set up earlier in the session. Most of the page is the same regardless of the user, but each user sees a customized *headline*—one that best fits the user's personal motivation for buying. (We are, after all, trying to sell him a car and become obscenely wealthy.) At the beginning of the session, a form asks the user to choose what's most important...

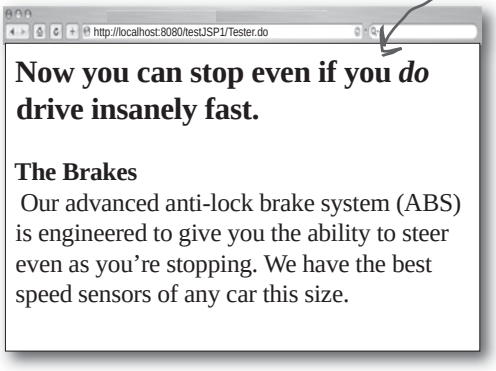
Imagine a web site for a car company. The first page asks the user what he feels is most important.

Just like a good salesman, the pages that talk about features of the car will customize the presentation based on the user's preference, so that each feature of the car looks like it was made with HIS personal needs in mind...

At the beginning of the session:



Somewhere later in the session:



The user's page is customized a little, to fit his interests...

The <c:if> tag won't work for this

There's no way to do exactly what we want using the <c:if> tag, because ***it doesn't have an "else".*** We can almost do it, using something like:

JSP using <c:if>, but it doesn't work right...

```
<c:if test="${userPref=='performance'}" >
    Now you can stop even if you <em>do</em> drive insanely fast..
</c:if>
<c:if test="${userPref=='safety'}" >
    Our brakes won't lock up no matter how bad a driver you are.
</c:if>
<c:if test="${userPref=='maintenance'}" >
    Lost your tech job? No problem--you won't have to service these brakes
    for at least three years.
</c:if>
```

But what happens if userPref doesn't match any of these? There's no way to specify the default headline?

```
<!-- continue with the rest of the page that EVERYONE should see  -->
```

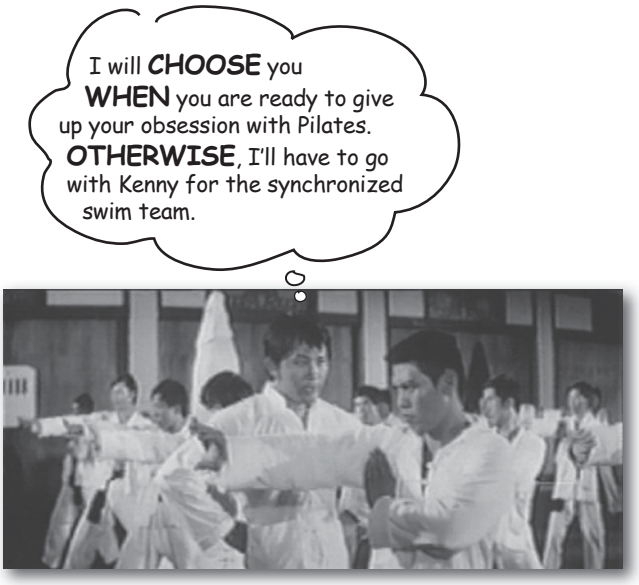
The <c:if> won't work unless we're CERTAIN that we'll never need a default value. What we really need is kind of an if/else construct:*

JSP *with* scripting, and it does what we want

Assume "userPref" was set somewhere earlier in the session.

```
<html><body><h2>
<% String pref = (String) session.getAttribute("userPref");
   if (pref.equals("performance")) {
       out.println("Now you can stop even if you <em>do</em> drive insanely fast.");
   } else if (pref.equals("safety")) {
       out.println("Our brakes won't lock up, no matter how bad a driver you are. ");
   } else if (pref.equals("maintenance")) {
       out.println(" Lost your tech job? No problem--you won't have to service these
brakes for at least three years.");
   } else {
       // userPref doesn't match those, so print the default headline
       out.println("Our brakes are the best.");
  } %>
</h2><strong>The Brakes</strong> <br>
Our advanced anti-lock brake system (ABS) is engineered to give you the ability to
steer even as you're stopping. We have the
best speed sensors of any car this size. <br>
</body></html>
```

*Yes, we agree with you—there's nearly *always* a better approach than chained if tests. But you're just gonna have to suspend disbelief long enough to learn how this all works....



The <c:choose> tag and its partners <c:when> and <c:otherwise>

No more than ONE of these four bodies (including the <c:otherwise>) will run.

(It's not like a switch statement--there's no fall-through.)

```
<c:choose>
    <c:when test="${userPref == 'performance'}">
        Now you can stop even if you <em>do</em> drive insanely fast.
    </c:when>

    <c:when test="${userPref == 'safety'}">
        Our brakes will never lock up, no matter how bad a driver you are.
    </c:when>

    <c:when test="${userPref == 'maintenance'}">
        Lost your tech job? No problem--you won't have to service these brakes
for at least three years.
    </c:when>

    <c:otherwise>
        Our brakes are the best.
    </c:otherwise>
</c:choose>
<!-- the rest of the page goes here... -->
```

If none of the <c:when> tests are true, the <c:otherwise> runs as a default.

Note: the <c:choose> tag is NOT required to have a <c:otherwise> tag.

The <c:set> tag... so much cooler than <jsp:setProperty>

The <jsp:setProperty> tag can do only one thing—set the property of a bean.

But what if you want to set a value in a Map? What if you want to make a *new* entry in a Map? Or what if you simply want to create a new request-scoped attribute?

You get all that with <c:set>, but you have to learn a few simple rules. Set comes in two flavors: ***var*** and ***target***. The *var* version is for setting attribute variables, the *target* version is for setting bean properties or Map values. Each of the two flavors comes in two variations: with or without a body. The <c:set> body is just another way to put in the *value*.

Setting an attribute variable *var* with <c:set>

1 With NO body

If there's NOT a session-scoped attribute named "userLevel", this tag creates one (assuming the value attribute is not null).

```
<c:set var="userLevel" scope="session" value="Cowboy" />
```

The scope is optional; var is required. You MUST specify a value, but you have a choice between putting in a value attribute or putting the value in the tag body (see #2 below).

value doesn't have to be a String...

```
<c:set var="Fido" value="${person.dog}" />
```

If ${person.dog} evaluates to a Dog object, then "Fido" is of type Dog.

2 WITH a body

Remember, no slash here when the tag has a body.

```
<c:set var="userLevel" scope="session" >
     Sheriff, Bartender, Cowgirl
</c:set>
```

The body is evaluated and used as the value of the variable.

If the *value* evaluates to null, the variable will be REMOVED! That's right, *removed*.

*Imagine that for the value (either in the body of the tag or using the value attribute), you use ${person.dog}. If ${person.dog} evaluates to null (meaning there is no **person**, or person's **dog** property is null, then if there IS a variable attribute with a name "Fido", that attribute will be removed! (If you don't specify a scope, it will start looking at page, then request, etc.). This happens even if the "Fido" attribute was originally set as a String, or a Duck, or a Broccoli.*

Using <c:set> with beans and Maps

This flavor of <c:set> (with its two variations—with and without a body) works for only two things: bean properties and Map values. That's it. You can't use it to add things to lists or arrays. It's simple—you give it the object (a bean or Map), the property/key name, and the value.

Setting a target property or value with <c:set>

1 With NO body

```
<c:set target="${PetMap}" property="dogName" value="Clover" />
```

If target is a bean, set the value of the property "dogName".

target must NOT be null!!

If target is a Map, set the value of a key named "dogName".

2 WITH a body

```
<c:set target="${person}" property="name" >
  ${foo.name}
</c:set>
```

Don't put the "id" name of the attribute here!

No slash... watch for this on the exam.

The body can be a String or expression.

The "target" must evaluate to the OBJECT! You don't type in the String "id" name of the bean or Map attribute!

This is a huge gotcha. In the <c:set> tag, the "target" attribute in the tag seems like it should work just like "id" in the <jsp:useBean>. Even the "var" attribute in the other version of <c:set> takes a String literal that represents the name of the scoped attribute. BUT... it doesn't work this way with "target"!
With the "target" attribute, you do NOT type in the String literal that represents the name under which the attribute was bound to the page, scope, etc. No, the "target" attribute needs a value that resolves to the REAL THING. That means an EL expression or a scripting expression (<%= %>), or something we haven't seen yet: <jsp:attribute>.

Key points and gotchas with <c:set>

Yes, <c:set> is easy to use, but there are a few deal-breakers you have to remember...

- You can never have BOTH the "var" and "target" attributes in a <c:set>.
- "Scope" is optional, but if you don't use it the default is *page* scope.
- If the "value" is null, the attribute named by "var" will be removed!
- If the attribute named by "var" does not exist, it'll be created, but only if "value" is not null.
- If the "target" expression is null, the Container throws an exception.
- The "target" is for putting in an expression that resolves to the Real Object. If you put in a String literal that represents the "id" name of the bean or Map, it won't work. In other words, "target" is not for the attribute *name* of the bean or Map—it's for the actual attribute *object*.
- If the "target" expression is not a Map or a bean, the Container throws an exception.
- If the "target" expression is a bean, but the bean does not have a property that matches "property", the Container throws an exception. Remember that the EL expression ${bean.notAProperty} will also throw an exception.

there are no Dumb Questions

Q: Why would I use the body version instead of the no-body version? It looks like they both do exactly the same thing.

A: That's because they DO... do the same thing. The body version is just for convenience when you want more room for the value. It might be a long and complex expression, for example, and putting it in the body makes it easier to read.

Q: If I don't specify a scope, does that mean it will find attributes that are ONLY within page scope, or does it do a search beginning with page scope?

A: If you don't use the optional "scope" attribute in the tag, then the tag will only look in the page scope space. Sorry, you will just have to know exactly which scope you are dealing with.

Q: Why is the word "attribute" so overloaded? It means both "the things that go inside tags" and "the things that are bound to objects in one of the four scopes." So you end up with an attribute of a tag whose value is an attribute of the page and...

A: We hear you. But that's what they're called. Once again, nobody asked US. We would have called the bound objects something like, oh, "bound objects".

<c:remove> just makes sense

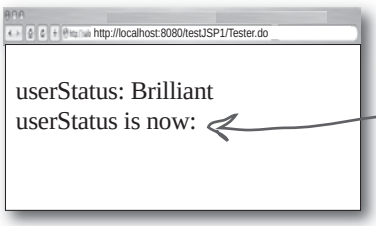
We agree with Dick—using a *set* to *remove* something feels wrong. (But remember, *set* does a *remove* only when you pass in a null value.)

The <c:remove> tag is intuitive and simple:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<html><body>

   <c:set var="userStatus" scope="request" value="Brilliant" />

   userStatus: ${userStatus} <br>

   <c:remove var="userStatus" scope="request" />

   userStatus is now: ${userStatus}

</body></html>
```

The var attribute MUST be a String literal! It can't be an expression!!

The scope is optional, but if you leave it out then the attribute is removed from ALL scopes.



The value of userStatus was removed, so nothing prints when the EL expression is used AFTER the remove.

Test your Tag memory

If you're studying for the exam, don't skip this one.
The answers are at the end of the chapter.

1. Fill in the name of the optional attribute.

```
<c:forEach var="movie" items="${movieList}" [          ]="foo"  >
   ${movie}
</c:forEach>
```

2. Fill in the missing attribute name.

```
<c:if [          ]="${userPref=='safety'}" >
    Maybe you should just walk...
</c:if>
```

3. Fill in the missing attribute name.

```
<c:set var="userLevel" scope="session"    [          ]="foo" />
```

4. Fill in the missing tag names (two different tag types), and the missing attribute name.

```
<c:choose>
    <c:[          ] [          ]="${userPref == 'performance'}">
        Now you can stop even if you <em>do</em> drive insanely fast.
    </c:[          ]>

    <c:[                        ] >
        Our brakes are the best.
    </c:[                        ]>
</c:choose>
```

With <c:import>, there are now THREE ways to include content

So far, we've used two different ways to add content from another resource into a JSP. But there's yet *another* way, using JSTL.

1 The include *directive*

```
<%@ include file="Header.html" %>
```

Static: adds the content from the value of the *file* attribute to the current page at ***translation*** time.

2 The <jsp:include> *standard action*

```
<jsp:include page="Header.jsp" />
```

Dynamic: adds the content from the value of the *page* attribute to the current page at ***request*** time.

3 The <c:import> JSTL tag

Unlike the other two includes, the <c:import> url can be from outside the web Container!

```
<c:import url="http://www.wickedlysmart.com/skyler/horse.html" />
```

Dynamic: adds the content from the value of the *URL* attribute to the current page, at ***request*** time. It works a lot like <jsp:include>, but it's more powerful and flexible.

Do NOT confuse <c:import> (a type of include) with the "import" attribute of the page directive (a way to put a Java import statement in the generated servlet).

They all have different attribute names!
(And watch out for "include" vs. "import")

*Each of the three mechanisms for including content from another resource into your JSP uses a different word for the attribute. The include directive uses **file**, the <jsp:include> uses **page**, and the JSTL <c:import> tag uses **url**. This makes sense, when you think about it... but you do have to memorize all three. The directive was originally intended for static layout templates, like HTML headers. In other words, a "file". The <jsp:include> was intended more for dynamic content coming from JSPs, so they named the attribute "page" to reflect that. The attribute for <c:import> is named for exactly what you give it—a URL! Remember, the first two "includes" can't go outside the current Container, but <c:import> **can**.*

<c:import> can reach OUTSIDE the web app

With <jsp:include> or the include directive, you can include only pages that are part of the current web app. But now with <c:import>, you have the option to pull in content from *outside* the Container. This simple example shows a JSP on Server A importing the contents of a URL on Server B. At request time, the HTML chunk in the imported file is added to the JSP. The imported chunk uses a reference to an image that is *also* on Server B.

Server A, the JSP doing the import

The JSP

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<html><body>
  <c:import url="http://www.wickedlysmart.com/skyler/horse.html" />
  <br>
  This is my horse.
</body></html>
```

(Don't forget: as with other include mechanisms, the thing you import should be an HTML fragment and NOT a complete page with opening and closing <html><body> tags.)

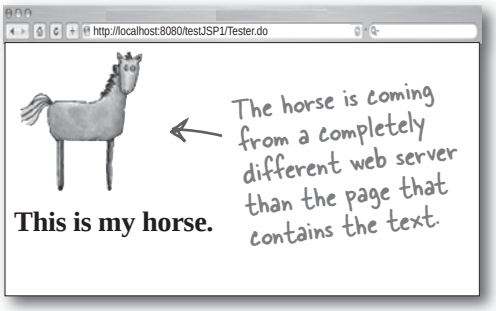
Server B, the imported content

The imported file

```
<img src="http://www.wickedlysmart.com/skyler/horse.gif">
```

"horse.html" and "horse.gif" are both on Server B, a completely different web server from the one with the JSP.

The response



Customizing the thing you include

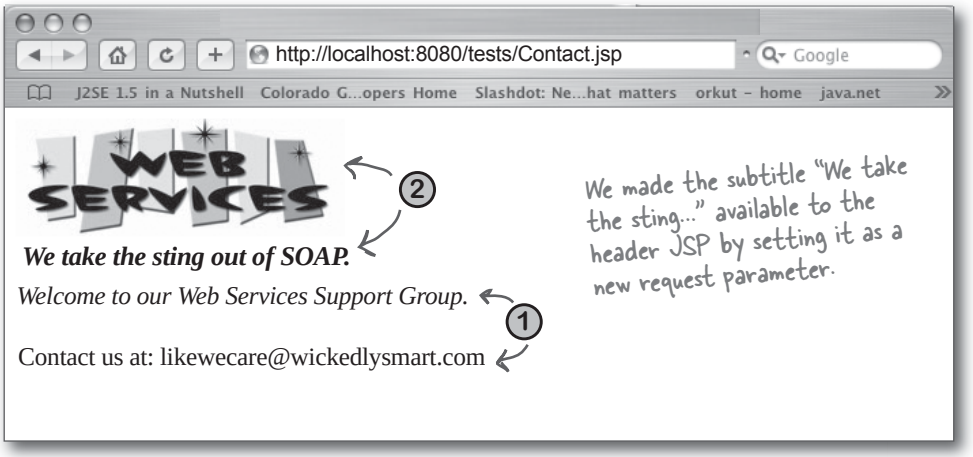
Remember in the previous chapter when we did a <jsp:include> to put in the layout header (a graphic with some text), but we wanted to customize the subtitle used in the header? We used <jsp:param> to make that happen...

1 The JSP with the <jsp:include>

```
<html><body>

<jsp:include page="Header.jsp">

   <jsp:param name="subTitle" value="We take the sting out of SOAP." />

</jsp:include>

<br>
<em>Welcome to our Web Services Support Group.</em> <br><br>
Contact us at: ${initParam.mainEmail}
</body></html>
```

2 The included file ("Header.jsp")

```
<img src="images/Web-Services.jpg" > <br>

<em><strong>${param.subTitle}</strong></em>
<br>
```



Doing the same thing with <c:param>

Here we accomplish the same thing we did on the previous page, but using a combination of <c:import> and <c:param>. You'll see that the structure is virtually identical to the one we used with standard actions.

1 The JSP with the <jsp:import>

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<html><body>

<c:import url="Header.jsp" >

   <c:param name="subTitle" value="We take the sting out of SOAP." />

</c:import>

<br>
<em>Welcome to our Web Services Support Group.</em> <br><br>

Contact us at: ${initParam.mainEmail}

</body></html>
```

No slash, because NOW the tag has a body...

2 The included file ("Header.jsp")

```
<img src="images/Web-Services.jpg" > <br>

<em><strong>${param.subTitle}</strong></em>
<br>
```

This page doesn't change at all. It doesn't care HOW the parameter got there, as long as it's there.

Sorry to change the subject here... but I just noticed a HUGE problem with JSPs! How can you guarantee session tracking from a JSP... without using scripting?

Session tracking happens automatically with JSPs, unless you explicitly disable it with a page directive that has a session attribute that says session="false".

He missed the point... I said "guarantee". My real question is--if the client doesn't support cookies, how can I get URL rewriting to happen? How can I get the session ID added to the URLs in my JSP?

Ahhh... he obviously doesn't know about the **<c:url>** tag. It does URL rewriting automatically.

<c:url> for all your hyperlink needs

Remember way back in our old servlet days when we wanted to use a session? First we had to *get* the session (either the existing one or a new one). At that point, the Container knows that it's supposed to associate the client from this request with a particular session ID. The Container *wants* to use a cookie—it wants to include a unique cookie with the response, and then the client will send that cookie back with each subsequent request. Except one problem... the client might have a browser with cookies disabled. Then what?

The Container will, automatically, fall back to URL rewriting if it doesn't get a cookie from the client. But with servlets, you STILL have to encode your URLs. In other words, *you* still have to tell the Container to "append the jsessionid to the end of this particular URL..." for each URL where it matters. Well, you can do the same thing from a JSP, using the <c:url> tag.

URL rewriting from a servlet

```
public void doGet(HttpServletRequest request, HttpServletResponse response)
                                                 throws IOException, ServletException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    HttpSession session = request.getSession();

    out.println("<html><body>");
    out.println("<a href=\"" + response.encodeURL("/BeerTest.do") + "\">click</a>");
    out.println("</body></html>");
}
```

Add the extra session ID info to this URL.

URL rewriting from a JSP

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<html><body>

This is a hyperlink with URL rewriting enabled.


<a href="<c:url value='/inputComments.jsp' />">Click here</a>


</body></html>
```

This adds the jsessionid to the end of the "value" relative URL (if cookies are disabled).

What if the URL needs encoding?

Remember that in an HTTP GET request, the parameters are appended to the URL as a query string. For example, if a form on an HTML page has two text fields—first name and last name—the request URL will stick the parameter names and values on to the end of the request URL. But...an HTTP request won't work correctly if it contains *unsafe* characters (although most modern browsers will try to compensate for this).

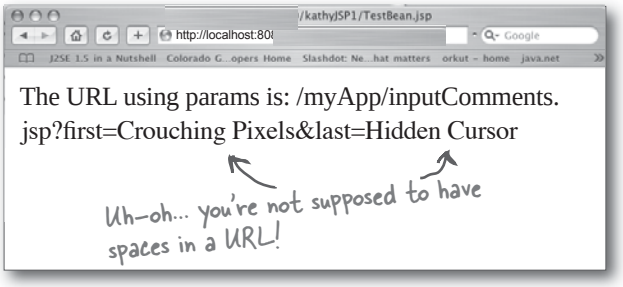
If you're a web developer, this is old news, but if you're new to web development, you need to know that URLs often need to be *encoded*. URL encoding means replacing the unsafe/reserved characters with other characters, and then the whole thing is decoded again on the server side. For example, spaces aren't allowed in a URL, but you can substitute a plus sign "+" for the space. The problem is, <c:url> does NOT automatically encode your URLs!

Using <c:url> with a query string

Remember, the <c:url> tag does URL rewriting, but *not* URL encoding!

```
<c:set var="last" value="Hidden Cursor" />
<c:set var="first" value="Crouching Pixels"/>

<c:url value="/inputComments.jsp?first=${first}&last=${last}" var="inputURL" />

The URL using params is: ${inputURL} <br>
```

Use the optional "var" attribute when you want access to this value later...



Yikes! Query string parameters have to be encoded... spaces, for example, must be replaced with a plus "+" sign.

Using <c:param> in the body of <c:url>

This solves our problem! Now we get both URL rewriting and URL encoding.

```
<c:url value="/inputComments.jsp" var="inputURL" >
   <c:param name="firstName" value="${first}" />
   <c:param name="lastName" value="${last}" />
</c:url>
```

no slash

Now we're safe, because <c:param> takes care of the encoding!

Now the URL looks like this:

```
/myApp/inputComments.jsp?firstName=Crouching+Pixels&lastName=Hidden+Cursor
```

I'm interrupting this JSTL talk for a few moments to talk about your error-handling. We're about to do something that might cause an exception...

You do NOT want your clients to see this:

Apache Tomcat/5.0.19 – Error report

http://localhost:8080/kathyJSP1/ChooseTest.jsp

J2SE 1.5 in a Nutshell Colorado G...opers Home Slashdot: Ne...hat matters orkut home java.net

HTTP Status 500 -

type Exception report

message

description The server encountered an internal error () that prevented it from fulfilling this request.

exception

```
org.apache.jasper.JasperException: / by zero
        org.apache.jasper.servlet.JspServletWrapper.service(JspServletWrapper.java:358)
        org.apache.jasper.servlet.JspServlet.serviceJspFile(JspServlet.java:301)
        org.apache.jasper.servlet.JspServlet.service(JspServlet.java:248)
        javax.servlet.http.HttpServlet.service(HttpServlet.java:856)
```

root cause

```
java.lang.ArithmeticException: / by zero
        org.apache.jsp.ChooseTest_jsp._jspService(ChooseTest_jsp.java:62)
        org.apache.jasper.runtime.HttpJspBase.service(HttpJspBase.java:133)
        javax.servlet.http.HttpServlet.service(HttpServlet.java:856)
        org.apache.jasper.servlet.JspServletWrapper.service(JspServletWrapper.java:311)
        org.apache.jasper.servlet.JspServlet.serviceJspFile(JspServlet.java:301)
        org.apache.jasper.servlet.JspServlet.service(JspServlet.java:248)
        javax.servlet.http.HttpServlet.service(HttpServlet.java:856)
```

note The full stack trace of the root cause is available in the Tomcat logs.

Apache Tomcat/5.0.19

Make your own error pages

The guy surfing your site doesn't want to see your stack trace. And he's not too thrilled to get a standard "404 Not Found", either.

You can't prevent *all* errors, of course, but you can at least give the user a friendlier (and more attractive) error response page. You can design a custom page to handle errors, then use the page directive to configure it.

The designated ERROR page ("errorPage.jsp")

```
<%@ page isErrorPage="true" %>

<html><body>
<strong>Bummer.</strong>
<img src="images/bummerGuy.jpg">
</body></html>
```

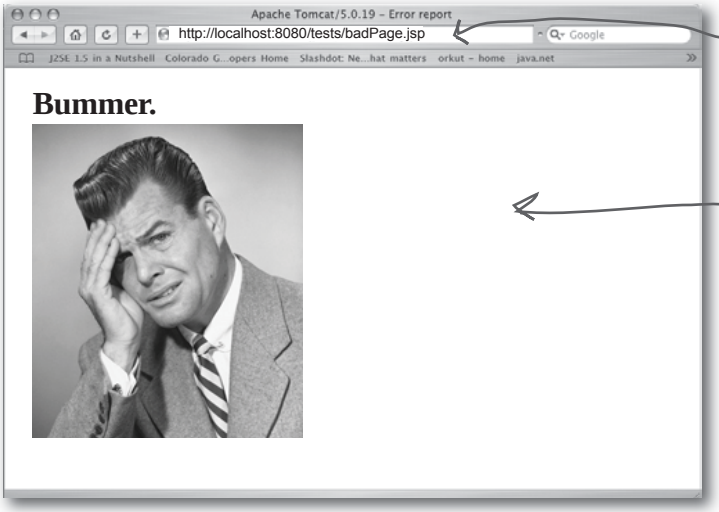
Confirms for the Container, "Yes, this IS an officially-designated error page."

The BAD page that throws an exception ("badPage.jsp")

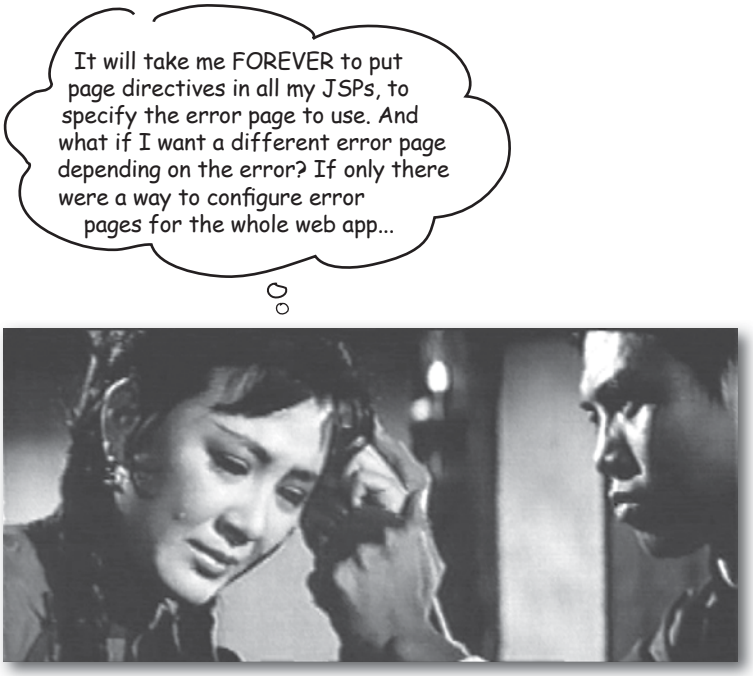
```
<%@ page errorPage="errorPage.jsp" %>

<html><body>
About to be bad...
<% int x = 10/0; %>
</body></html>
```

Tells the Container, "If something goes wrong here, forward the request to errorPage.jsp".

What happens when you request "badPage.jsp"



The REQUEST was for "badPage.jsp", but that page threw an exception, so the RESPONSE came from "errorPage.jsp".



She doesn't know about the <error-page> DD tag.

You can declare error pages in the DD for the entire web app, and you can even configure *different* error pages for different exception types, or HTTP error code types (404, 500, etc.).

The Container uses <error-page> configuration in the DD as the default, but if a JSP has an explicit *errorPage* page directive, the Container uses the directive.

Configuring error pages in the DD

You can declare error pages in the DD based on either the <exception-type> or the HTTP status <error-code> number. That way you can show the client different error pages specific to the type of the problem that generated the error.

Declaring a catch-all error page

This applies to everything in your web app—not just JSPs. You can override it in individual JSPs by adding a page directive with an *errorPage* attribute.

```
<error-page>
  <exception-type>java.lang.Throwable</exception-type>
  <location>/errorPage.jsp</location>
</error-page>
```

Declaring an error page for a more explicit exception

This configures an error page that's called only when there's an ArithmeticException. If you have both this declaration and the catch-all above, any exception other than ArithmeticException will still end up at the "errorPage.jsp".

```
<error-page>
  <exception-type>java.lang.ArithmeticException</exception-type>
  <location>/arithmeticError.jsp</location>
</error-page>
```

Declaring an error page based on an HTTP status code

This configures an error page that's called only when the status code for the response is "404" (file not found).

```
<error-page>
  <error-code>404</error-code>
  <location>/notFoundError.jsp</location>
</error-page>
```

The <location> MUST be relative to the web-app root/context, which means it MUST start with a slash. (This is true regardless of whether the error page is based on <error-code> or <exception-type>.)

Error pages get an extra object: exception

An error page is essentially the JSP that *handles* the exception, so the Container gives the page an extra object for the *exception*. You probably won't want to show the exception to the user, but you've got it. In a scriptlet, you can use the implicit object *exception*, and from a JSP, you can use the EL implicit object ${pageContext.exception}. The object is type java.lang.Throwable, so in a script you can call methods, and with EL you can access the *stackTrace* and *message* properties.

Note: the exception implicit object is available ONLY to error pages with an explicitly-defined page directive:

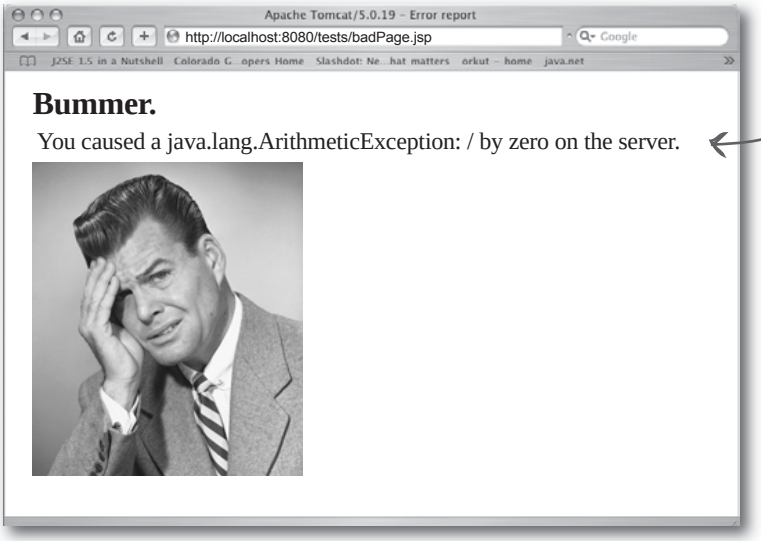
<%@ page isErrorPage="true" %>

In other words, configuring an error page in the DD is not enough to make the Container give that page the implicit exception object!

A more explicit ERROR page ("errorPage.jsp")

```
<%@ page  isErrorPage="true" %>

<html><body>
<strong>Bummer.</strong><br>

You caused a ${pageContext.exception} on the server.<br>

<img src="images/bummerGuy.jpg">
</body></html>
```

What happens when you request "badPage.jsp"



This time, you get more details. You probably won't show this to the user...we just did this so you could see it.

What if I think there's an exception I might be able to recover from in a JSP? What if there are some errors I want to catch myself?

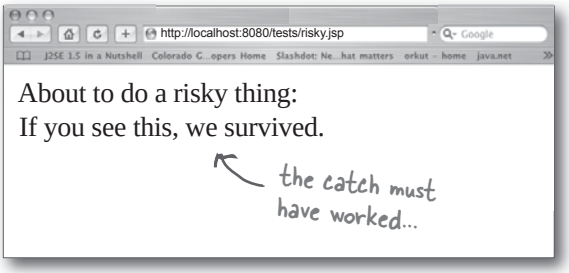
The <c:catch> tag. Like try/catch...*sort of*

If you have a page that invokes a risky tag, but you think you can recover, there's a solution. You can do a kind of try/catch using the <c:catch> tag, to wrap the risky tag or expression. Because if you don't, and an exception is thrown, your default error handling will kick in and the user will get the error page declared in the DD. The part that might feel a little strange is that the <c:catch> serves as both the try *and* the catch—there's no separate *try* tag. You wrap the risky EL or tag calls or whatever in the body of a <c:catch>, and the exception is caught right there. But you can't assume it's exactly like a catch block, either, because once the exception occurs, control jumps to the end of the <c:catch> tag body (more on that in a minute).

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ page errorPage="errorPage.jsp" %>
<html><body>

About to do a risky thing: <br>

<c:catch>

    <% int x = 10/0; %>

</c:catch>

If you see this, we survived.

</body></html>
```

This scriptlet will DEFINITELY cause an exception... but we caught it instead of triggering the error page.

If this prints out, then we KNOW we made it past the exception (which in this example, means we successfully caught the exception).



But how do I get access to the Exception object? The one that was actually thrown? Since this isn't an actual error page, the implicit exception object doesn't work here.

You can make the exception an attribute

In a real Java try/catch, the catch argument is the exception object. But with web app error handling, remember, *only officially-designated error pages get the exception object.* To any other page, the exception just isn't there. So this does *not* work:

```
<c:catch>
    Inside the catch...
    <% int x = 10/0; %>
</c:catch>

Exception was: ${pageContext.exception}
```

Won't work because this isn't an official error page, so it doesn't get the exception object.

Using the "var" attribute in <c:catch>

Use the optional *var* attribute if you want to access the exception after the end of the <c:catch> tag. It puts the exception object into the page scope, under the name *you* declare as the value of *var*.

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ page errorPage="errorPage.jsp" %>
<html><body>

About to do a risky thing: <br>

<c:catch var="myException">

    Inside the catch...
    <% int x = 10/0; %>
</c:catch>

<c:if test="${myException != null}">
   There was an exception:  ${myException.message} <br>
</c:if>

We survived.
</body></html>
```

This creates a new page-scoped attribute named "myException", and assigns the exception object to it.

Now there's an attribute myException, and since it's a Throwable, it has a "message" property (because Throwable has a getMessage() method).

Flow control works in a <c:catch> the way it does in a *try* block—NOTHING runs inside the <c:catch> body *after* the exception.

In a regular Java try/catch, once the exception occurs, the code BELOW that point in the ***try*** *block never executes—control jumps directly to the catch block. With the <c:catch> tag, once the exception occurs, two things happen:*

1) If you used the optional "var" attribute, the exception object is assigned to it.

2) Flow jumps to ***below*** *the body of the <c:catch> tag.*

```
<c:catch>

    Inside the catch...
    <% int x = 10/0; %>
    After the catch...

</c:catch>

We survived.
```

control

You'll NEVER see this!

Be careful about this. If you want to use the "var" exception object, you must wait until AFTER you get to the end of the <c:catch> body. In other words, there is simply no way to use any information about the exception WITHIN the <c:catch> tag body.

It's tempting to think of a <c:catch> tag as being just like a normal Java code catch block, but it isn't. A <c:catch> acts more like a ***try*** *block, because it's where you put the risky code. Except it's like a try that never needs (or has) a catch or finally block. Confused? The point is—learn this tag for exactly what it is, rather than mapping it into your existing knowledge of how a normal try/catch works. And on the exam, if you see code within the <c:catch> tag that is below the point at which the exception is thrown, don't be fooled.*

What if you need a tag that's NOT in JSTL?

The JSTL is huge. Version 1.1 has *five* libraries—four with custom *tags*, and one with a bunch of *functions* for String manipulation. The tags we cover in this book (which happen to be the ones you're expected to know for the exam) are for the generic things you're most likely to need, but it's possible that between all five libraries, you'll find everything you might ever need. On the next page, we'll start looking at what happens when the tags below aren't enough.

The "Core" library

General-purpose

```
<c:out>
<c:set>
<c:remove>
<c:catch>
```

Conditional

```
<c:if>
<c:choose>
<c:when>
<c:otherwise>
```

URL related

```
<c:import>
<c:url>
<c:redirect>
<c:param>
```

Iteration

```
<c:forEach>
<c:forTokens>
```

We didn't cover this one... it lets you iterate over tokens where YOU give it the delimiter. Works a lot like StringTokenizer. We also didn't cover <c:redirect> and <c:out>, but that gives you a wonderful excuse to get the JSTL docs.

The "Formatting" library

Internationalization

```
<fmt:message>
<fmt:setLocale>
<fmt:bundle>
<fmt:setBundle>
<fmt:param>
<fmt:requestEncoding>
```

Formatting

```
<fmt:timeZone>
<fmt:setTimeZone>
<fmt:formatNumber>
<fmt:parseNumber>
<fmt:parseDate>
```

The "SQL" library

Database access

```
<sql:query>
<sql:update>
<sql:setDataSource>
<sql:param>
<sql:dateParam>
```

The "XML" library

Core XML actions

```
<x:parse>
<x:out>
<x:set>
```

XML flow control

```
<x:if>
<x:choose>
<x:when>
<x:otherwise>
<x:forEach>
```

Transform actions

```
<x:transform>
<x:param>
```

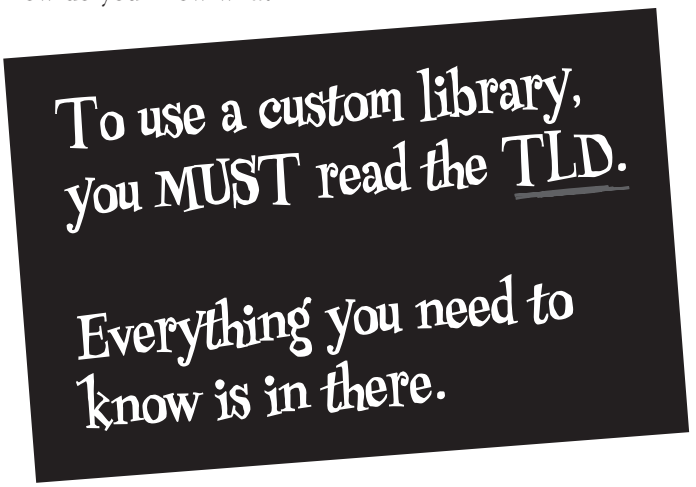
Only the "core" library is covered on the exam.

The "core" library (which by convention we always prefix with "c") is the only JSTL library covered on the exam. The rest are specialized, so we don't go into them. But you should at least know that they're available. The XML transformation tags, for example, could save your life if you have to process RSS feeds. Writing your own custom tags can be a pain, so make sure before you write one that you're not reinventing the wheel.

Using a tag library that's NOT from the JSTL

Creating the code that goes *behind* a tag (in other words, the Java code that's invoked when you put the tag in your JSP) isn't trivial. We have a whole chapter (the next one) devoted to developing your own custom tag handlers. But the last part of this chapter is about how to *use* custom tags. What happens, for example, if someone hands you a custom tag library they created for your company or project? How do you know what the tags are and how to use them? With JSTL, it's easy—the JSTL 1.1 specification *documents* each tag, including how to use each of the required and optional attributes.

But not every custom tag will come so nicely packaged and well-documented. You have to know how to figure out a tag even if the documentation is weak or nonexistent, and, one more thing—you have to know how to *deploy* a custom tag library.



Main things you have to know:

1 The tag name and syntax

The tag has a *name*, obviously. In <c:set>, the tag *name* is *set*, and the *prefix* is *c*. You can use any prefix you want, but the *name* comes from the TLD. The syntax includes things like required and optional attributes, whether the tag can have a body (and if so, what you can put there), the type of each attribute, and whether the attribute can be an expression (vs. a literal String).

2 The library URI

The URI is a unique identifier in the Tag Library Descriptor (TLD). In other words, it's a unique name for the tag library the TLD describes. The URI is what you put in your taglib directive. It's what tells the Container how to identify the TLD file within the web app, which the Container needs in order to map the tag name used in the JSP to the Java code that runs when you use the tag.

Making sense of the TLD

The TLD describes two main things: custom tags, and EL functions. We used one when we made the dice rolling function in the previous chapter, but we had only a <function> element in the TLD. Now we have to look at the <tag> element, which can be more complex. Besides the function we declared earlier, the TLD below describes one tag, *advice*.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<taglib xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee/web-jsptaglibrary_2_0.xsd"
version="2.0">

    <tlib-version>1.2</tlib-version>

    <short-name>RandomTags</short-name>
    <function>
       <name>rollIt</name>
       <function-class>foo.DiceRoller</function-class>
       <function-signature>int rollDice()</function-signature>
    </function>

    <uri>randomThings</uri>

    <tag>

       <description>random advice</description>

       <name>advice</name>

       <tag-class>foo.AdvisorTagHandler</tag-class>

       <body-content>empty</body-content>

       <attribute>
          <name>user</name>

          <required>true</required>

          <rtexprvalue>true</rtexprvalue>

       </attribute>

    </tag>
</taglib>
```

This is the version of the XML schema that you use for JSP 2.0. Don't memorize it... just copy it into your <taglib> element.

MANDATORY (the tag, not the value)— the developer puts it in to declare the version of the tag library.

MANDATORY; mainly for tools to use..

The EL function we used in the last chapter.

The unique name we use in the taglib directive!

Optional, but a really good idea...

REQUIRED! This is what you use inside the tag (example: <my:advice>).

REQUIRED! This is how the Container knows what to call when someone uses the tag in a JSP.

REQUIRED! This says that the tag must NOT have anything in the body.

If your tag has attributes, then one <attribute> element per tag attribute is required.

This says you MUST put a "user" attribute in the tag.

This says the "user" attribute can be a runtime expression value (i.e. doesn't have to be a String literal).

Using the custom "advice" tag

The "advice" tag is a simple tag that takes one attribute—the user name—and prints out a piece of random advice. It's simple enough that it *could* have been just a plain old EL function (with a static method getAdvice(String name)), but we made it a simple tag to show you how it all works...

The TLD elements for the advice tag

```
<taglib ...>
...
<uri>randomThings</uri>
<tag>
    <description>random advice</description>
    <name>advice</name>
    <tag-class>foo.AdvisorTagHandler</tag-class>
    <body-content>empty</body-content>

   <attribute>
      <name>user</name>
      <required>true</required>
      <rtexprvalue>true</rtexprvalue>
   </attribute>

</tag>
</taglib ...>
```

This is the same tag you saw on the previous page, but without the annotations.

JSP that uses the tag

```
<html><body>

<%@ taglib prefix="mine" uri="randomThings"%>

Advisor Page<br>

<mine:advice user="${userName}" />

</body></html>
```

The uri matches the <uri> element in the TLD.

It's OK to use EL here, because the <rtexprevalue> in the TLD is set to "true" for the user attribute. (Assume the "userName" attribute already exists.)

The TLD says the tag can't have a body, so we made it an empty tag (which means the tag ends with a slash).

Each library you use in a page needs its own taglib directive with a unique prefix.

The custom tag handler

This simple tag handler extends SimpleTagSupport (a class you'll see in the next chapter), and implements two key methods: doTag(), the method that does the actual work, and setUser(), the method that accepts the attribute value.

Java class that does the tag work

```
package foo;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.tagext.SimpleTagSupport;
import java.io.IOException;

public class AdvisorTagHandler extends SimpleTagSupport {

    private String user;

    public void doTag() throws JspException, IOException {
       getJspContext().getOut().write( "Hello " + user + " <br>" );
       getJspContext().getOut().write( "Your advice is: " + getAdvice() );
    }

    public void setUser(String user) {
       this.user=user;
    }

    String getAdvice() {
       String[] adviceStrings = {"That color's not working for you.",
              "You should call in sick.", "You might want to rethink that haircut."};
       int random = (int) (Math.random() * adviceStrings.length);
       return adviceStrings[random];
    }
}
```

SimpleTagSupport implements things we need in custom tags.

The Container calls doTag() when the JSP invokes the tag using the name declared in the TLD.

The Container calls this method to set the value from the tag attribute. It uses JavaBean property naming conventions to figure out that a "user" attribute should be sent to the setUser() method.

Our own internal method.

Custom tag handlers don't use custom method names!

With EL functions, you created a Java class with a static method, named the method whatever you wanted, then used the TLD to map the actual method <function-signature> to the function <name>. But with custom tags, the method name is ALWAYS **doTag()**, *so you never declare the method name for a custom tag. Only functions use a method signature declaration in the TLD!*

Pay attention to <rtexprvalue>

The <rtexprvalue> is especially important because it tells you whether the value of the attribute is evaluated at translation or runtime. If the <rtexprvalue> is false, or the <rtexprvalue> isn't defined, you can use only a String literal as that attribute's value!

If you see this:

```
<attribute>
    <name>rate</name>
    <required>true</required>
    <rtexprvalue>false</rtexprvalue>
</attribute>
```

OR this:

```
<attribute>
    <name>rate</name>
    <required>true</required>

</attribute>
```

If there's no <rtexprvalue>, the default value is false.

Then you know THIS WON'T WORK!

```
<html><body>
  <%@ taglib prefix="my" uri="myTags"%>

  <my:handleIt rate="${currentRate}" />
</body></html>
```

NO! This must NOT be an expression... it must be a String literal.

Q: **You still didn't answer the question about how you know what type the attribute is...**

A: We'll start with the easy one. If the <rtexprvalue> is false (or not there at all), then the attribute type can be ONLY a String literal. But if you can use an expression, then you have to hope that it's either dead obvious from the tag description and attribute name, OR that the developer included the optional <type> subelement of the <attribute> element. The <type> takes a fully-qualified class name for the type. Whether the TLD declares the type or not, the Container expects the type of the expression to match the type of argument in the tag handler's setter method for that attribute. In other words, if the tag handler has a setDog(Dog) method for the "dog" attribute, then the value of your expression for that attribute better evaluate to a Dog object! (Or something that can be implicitly assigned to a Dog reference type.)

<rtexprvalue> is NOT just for EL expressions

You can use ***three*** kinds of expressions for the value of an attribute (or tag body) that allows runtime expressions.

1. EL expressions

```
<mine:advice user="${userName}" />
```

2. Scripting expressions

```
<mine:advice user='<%= request.getAttribute("username") %>' />
```

It has to be an expression, not just a scriplet. So it must have the "=" sign in there and no semicolon on the end.

3. <jsp:attribute> standard actions

```
<mine:advice>
   <jsp:attribute name="user">${userName}</jsp:attribute>
</mine:advice>
```

What is this?? I thought this tag didn't have a body...

<jsp:attribute> lets you put attributes in the BODY of a tag, even when the tag body is explicitly declared "empty" in the TLD!!

The <jsp:attribute> is simply an alternate way to define attributes to a tag. The key point is, there must be only ONE <jsp:attribute> for EACH attribute in the enclosing tag. So if you have a tag that normally takes three attributes IN the tag (as opposed to in the body), then inside the body you'll now have three <jsp:attribute> tags, one for each attribute. Also notice that the <jsp:attribute> has an attribute of its own, ***name****, where you specify the name of the outer tag's attribute for which you're setting a value.*
There's a little more about this on the next page...

What can be in a tag body

A tag can have a body *only* if the <body-content> element for this tag is not configured with a value of ***empty***. The <body-content> element can be one of either three or four values, depending on the type of tag.

```
<body-content>empty</body-content>
```
The tag must NOT have a body.

```
<body-content>scriptless</body-content>
```
The tag must NOT have scripting elements (scriptlets, scripting expressions, and declarations), but it CAN have template text and EL and custom and standard actions.

```
<body-content>tagdependent</body-content>
```
The tag body is treated as plain text, so the EL is NOT evaluated and tags/actions are not triggered.

```
<body-content>JSP</body-content>
```
The tag body can have anything that can go inside a JSP.

THREE ways to invoke a tag that can't have a body

Each of these are acceptable ways to invoke a tag configured in the TLD with <body-content>**empty**</body-content>.

1. An *empty* tag

```
<mine:advice user="${userName}" />
```
When you put a slash in the opening tag, you don't use a closing tag.

2. A tag with *nothing* between the opening and closing tags

```
<mine:advice user="${userName}"> </mine:advice>
```
We have an opening and closing tag, but NOTHING in between.

3. A tag with only <jsp:attribute> tags between the opening and closing tags

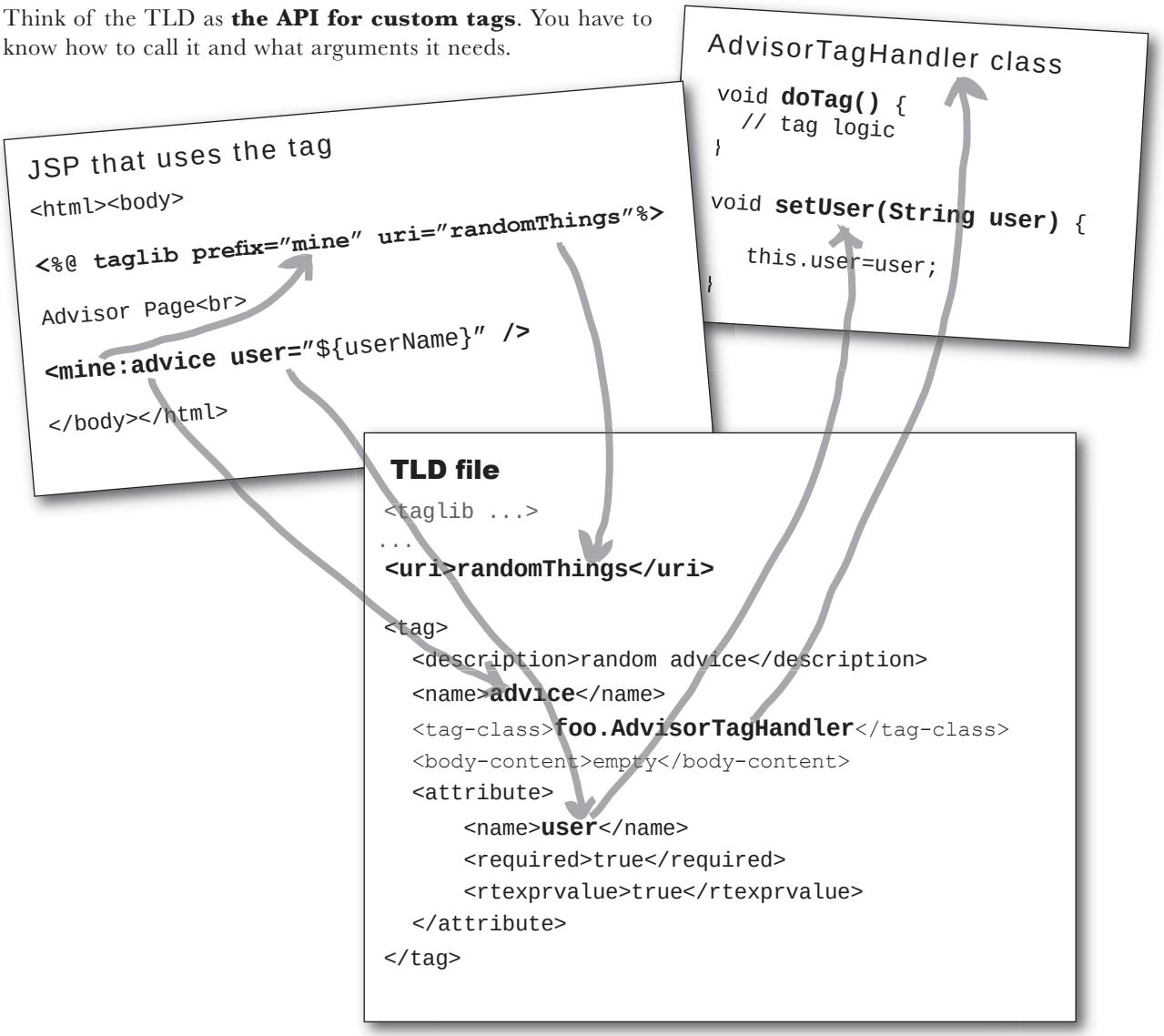
```
<mine:advice>
   <jsp:attribute name="user">${userName}</jsp:attribute>
</mine:advice>
```
The <jsp:attribute> tag is the ONLY thing you can put between the opening and closing tags of a tag with a <body-content> of empty! It's just an alternate way to put the attributes in, but <jsp:attribute> tags don't count as "body content".

The tag handler, the TLD, and the JSP

The tag handler developer creates the TLD to tell both the Container and the JSP developer how to use the tag. A JSP developer doesn't care about the <tag-class> element in the TLD; that's for the Container to worry about. The JSP developer cares most about the uri, the tag name, and the tag syntax. Can the tag have a body? Does this attribute have to be a String literal, or can it be an expression? Is this attribute optional? What type does the expression need to evaluate to?

Think of the TLD as **the API for custom tags**. You have to know how to call it and what arguments it needs.

These three pieces—the tag handler class, the TLD, and the JSP are all you need to deploy and run a web app that uses the tag.



The taglib <uri> is just a name, not a location

The <uri> element in the TLD is a unique name for the tag library. That's it. It does NOT need to represent any actual location (path or URL, for example). It simply has to be a name—*the same name you use in the taglib directive.*

"But," you're asking, "how come with the JSTL it gives the full URL to the library?" The taglib directive for the JSTL is:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
```

This LOOKS like a URL to a web resource, but it's not. It's just a name that happens to be formatted as a URL.

The web Container doesn't normally try to *request* something from the uri in the taglib directive. It doesn't need to use the uri as a *location*! If you type that as a URL into your browser, you'll be redirected to a different URL, one that has *information* about JSTL. The Container could care less that this particular uri happens to also be a valid URL (the whole "http://..." thing). It's just the convention Sun uses for the uri, to help ensure that it's a unique name. Su could have named the JSTL uri "java_foo_tags" and it would have worked in exactly the same way. ***All that matters is that the <uri> in the TLD and the uri in the taglib directive match!***

As a developer, though, you do want to work out a scheme to give your libraries unique <uri> values, because <uri> names need to be *unique* for any given web app. You can't, for example, have two TLD files in the same web app, with the same <uri>. So, the domain name convention is a good one, but you don't necessarily need to use that for all of your in-house development.

Having said all that, there *is* one way in which the uri could be used as a location, but it's considered a really bad practice—if you don't specify a <uri> inside the TLD, the Container will attempt to use the uri attribute in the taglib directive as a path to the actual TLD. But to hard-code the location of your TLD is obviously a bad idea, so just pretend you don't know it's possible.

The Container looks for a match between the <uri> in the TLD and the uri value in the taglib directive. The uri does NOT have to be the location of the actual tag handler!

The Container builds a map

Before JSP 2.0, the developer had to specify a mapping between the <uri> in the TLD and the actual location of the TLD file. So when a JSP page had a taglib directive like this:

```
<%@ taglib prefix="mine" uri="randomThings"%>
```

The Deployment Descriptor (web.xml) had to tell the Container where the TLD file with a matching <uri> was located. You did that with a <taglib> element in the DD.

The OLD (before JSP 2.0) way to map a taglib uri to a TLD file

```
<web-app>
...
  <jsp-config>
    <taglib>
      <taglib-uri>randomThings</taglib-uri>
      <taglib-location>/WEB-INF/myFunctions.tld</taglib-location>
    </taglib>
  </jsp-config>
</web-app>
```

In the DD, map the <uri> in the TLD to an actual path to a TLD file.

The NEW (JSP 2.0) way to map a taglib uri to a TLD file

No <taglib> entry in the DD!

The Container automatically builds a map between TLD files and <uri> names, so that when a JSP invokes a tag, the Container knows exactly where to find the TLD that describes the tag.

How? By looking through a specific set of locations where TLDs are allowed to live. When you deploy a web app, as long as you put the TLD in a place the Container will search, the Container will find the TLD and build a map for that tag library.

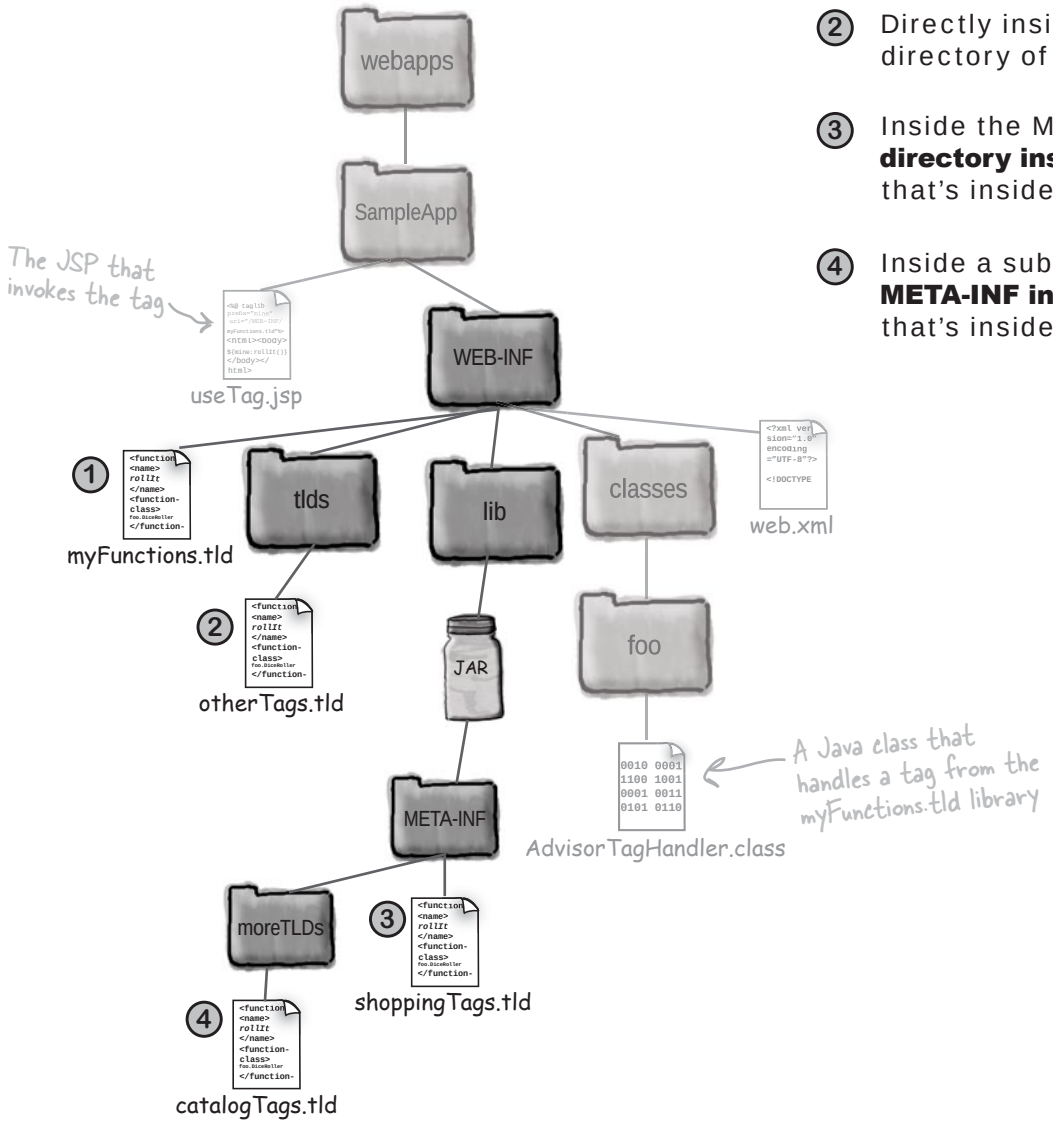
If you *do* specify an explicit <taglib-location> in the DD (web.xml), a JSP 2.0 Container will use it! In fact, when the Container begins to build the <uri>-to-TLD map, the Container will look *first* in your DD to see if you've made any <taglib> entries, and if you have, it'll use those to help construct the map. ***For the exam, you're expected to know about <taglib-location>, even though it's no longer required for JSP 2.0.***

So the next step is for us to see where the Container looks for TLDs, and also where it looks for the tag handler *classes* declared in the TLDs.

Four places the Container looks for TLDs

The Container searches in several places to find TLD files—you don't need to do anything except make sure your TLDs are in one of the right locations.

1. Directly inside WEB-INF
2. Directly inside a sub-directory of WEB-INF
3. Inside the META-INF **directory inside a JAR file** that's inside WEB-INF/lib
4. Inside a sub-directory of **META-INF inside a JAR file** that's inside WEB-INF/lib



When a JSP uses more than one tag library

If you want to use more than one tag library in a JSP, do a separate taglib directive for each TLD. There a few issues to keep in mind...

- Make sure the taglib uri names are unique. In other words, don't put in more than one directive with the same uri value.

- Do NOT use a prefix that's on the reserved list.
  The reserved prefixes are:

  jsp:

  jspx:

  java:

  javax:

  servlet:

  sun:

  sunw:

Sharpen your pencil

Empty tags

Write in examples of the THREE different ways to invoke a tag that must have an empty body.
(Check your answers by looking back through the chapter. No, we're not going to tell you the page number.)

1. _____

2. _____

3. _____

Sharpen your pencil

How the JSP, the TLD, and the bean attribute class relate

Fill in the spaces based on the information that you can see in the TLD. Draw arrows to indicate where the different pieces of information are tied together. In other words, for each blank, show exactly where you found the information needed to fill in the blank.

JSP that uses the tag

```
<html><body>

<%@ taglib prefix="mine" uri="_____"%>

Advisor Page<br>

<_____:_____ _____="${foo}" />

</body></html>
```

AdvisorTagHandler class

```
void doTag() {
  // tag logic
}

void set_____(String x) {

  // code here
}
```

TLD file

```
<taglib ...>
...
<uri>randomThings</uri>

<tag>
  <description>random advice</description>
  <name>advice</name>
  <tag-class>foo.AdvisorTagHandler</tag-class>
  <body-content>empty</body-content>
  <attribute>
      <name>user</name>
      <required>true</required>
      <rtexprvalue>_____</rtexprvalue>
  </attribute>
</tag>
```

Test your Tag memory ANSWERS

① Fill in the name of the optional attribute.

The attribute that names the loop counter variable.

```
<c:forEach var="movie" items="${movieList}" varStatus="foo"  >
   ${movie}
</c:forEach>
```

② Fill in the missing attribute name.

```
<c:if test="${userPref=='safety'}" >
    Maybe you should just walk...
</c:if>
```

The <c:set> tag must have a value, but you could choose to put the value in the body of the tag instead of as an attribute.

③ Fill in the missing attribute name.

```
<c:set var="userLevel" scope="session"  value="foo" />
```

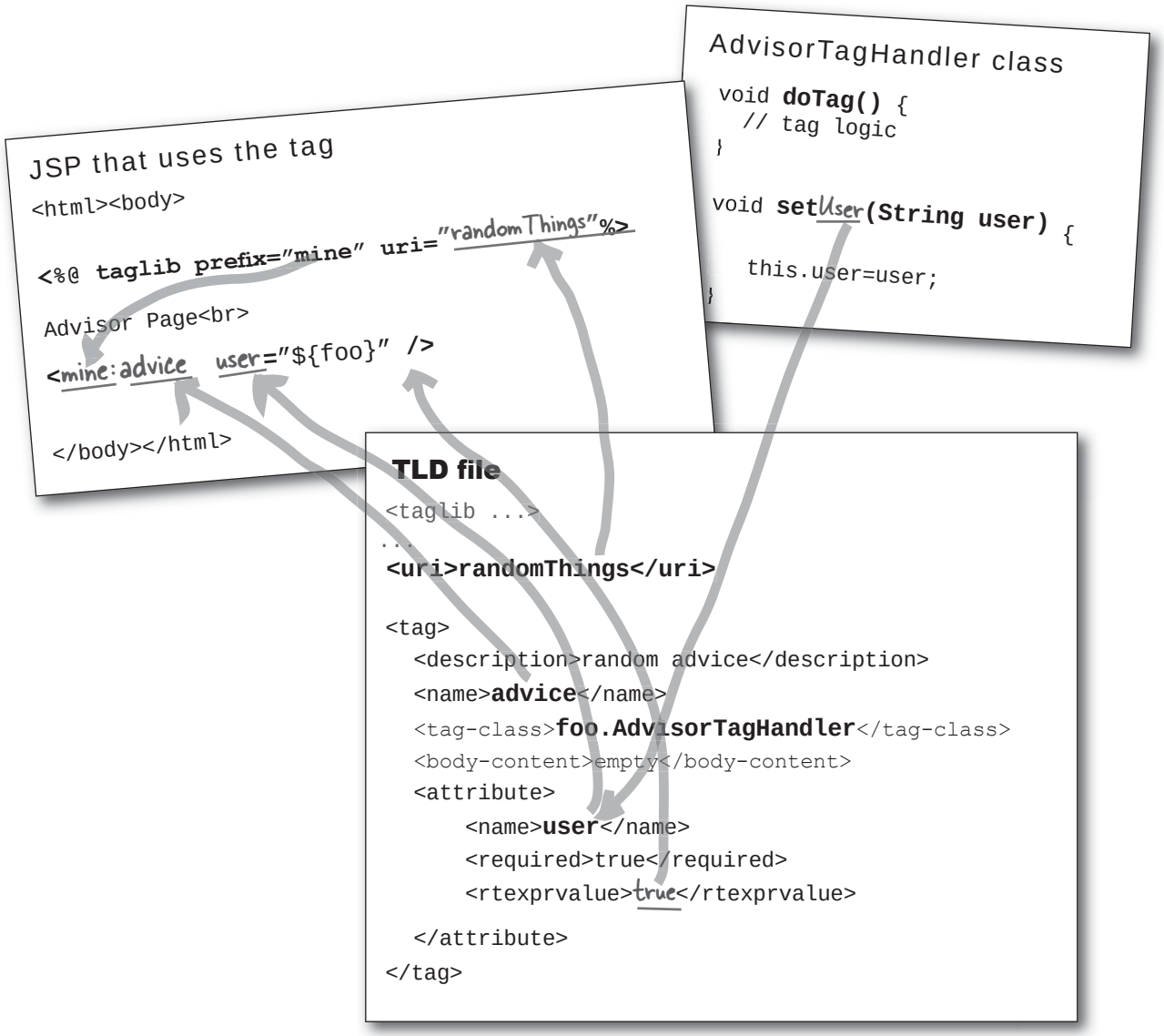
④ Fill in the missing tag names (two different tag types), and the missing attribute name.

```
<c:choose>
    <c:when test="${userPref == 'performance'}">
        Now you can stop even if you <em>do</em> drive insanely fast.
    </c:when>
    <c:otherwise >
        Our brakes are the best.
    </c:otherwise>
</c:choose>
```

The <c:otherwise> tag is optional.

Sharpen your pencil

How the JSP, the TLD, and the bean attribute class relate
ANSWERS



Mock Exam Chapter 9

1 Which is true about TLD files?

- ❑ A. TLD files may be placed in any subdirectory of **`WEB-INF`**.
- ❑ B. TLD files are used to configure JSP environment attributes, such as **`scripting-invalid`**.
- ❑ C. TLD files may be placed in the **`META-INF`** directory of the WAR file.
- ❑ D. TLD files can declare both Simple and Classic tags, but TLD files are NOT used to declare Tag Files.

2 Assuming the standard JSTL prefix conventions are used, which JSTL tags would you use to iterate over a collection of objects? (Choose all that apply.)

- ❑ A. **`<x:forEach>`**
- ❑ B. **`<c:iterate>`**
- ❑ C. **`<c:forEach>`**
- ❑ D. **`<c:forTokens>`**
- ❑ E. **`<logic:iterate>`**
- ❑ F. **`<logic:forEach>`**

3 A JSP page contains a **taglib** directive whose **uri** attribute has the value **myTags**. Which deployment descriptor element defines the associated TLD?

- ❑ A.
```
<taglib>
  <uri>myTags</uri>
  <location>/WEB-INF/myTags.tld</location>
</taglib>
```
- ❑ B.
```
<taglib>
  <uri>myTags</uri>
  <tld-location>/WEB-INF/myTags.tld</tld-location>
</taglib>
```
- ❑ C.
```
<taglib>
  <tld-uri>myTags</tld-uri>
  <tld-location>/WEB-INF/myTags.tld</tld-location>
</taglib>
```
- ❑ D.
```
<taglib>
  <taglib-uri>myTags</taglib-uri>
  <taglib-location>/WEB-INF/myTags.tld</taglib-location>
</taglib>
```

4 A JavaBean **Person** has a property called **address**. The value of this property is another JavaBean **Address** with the following string properties: **street1**, **street2**, **city**, **stateCode** and **zipCode**. A controller servlet creates a session-scoped attribute called **customer** that is an instance of the **Person** bean.

Which JSP code structures will set the **city** property of the **customer** attribute to the **city** request parameter? (Choose all that apply.)

- ❑ A.
```
${sessionScope.customer.address.city = param.city}
```
- ❑ B.
```
<c:set target="${sessionScope.customer.address}"
       property="city" value="${param.city}" />
```
- ❑ C.
```
<c:set scope="session" var="${customer.address}"
       property="city" value="${param.city}" />
```
- ❑ D.
```
<c:set target="${sessionScope.customer.address}"
       property="city">
  ${param.city}
</c:set>
```

5 Which **<body-content>** element combinations in the TLD are valid for the following JSP snippet? (Choose all that apply.)

```
11. <my:tag1>
12.   <my:tag2 a="47" />
13.   <% a = 420; %>
14.   <my:tag3>
15.     value = ${a}
16.   </my:tag3>
17. </my:tag1>
```

- ❑ A. tag1 body-content is **empty**
  tag2 body-content is **JSP**
  tag3 body-content is **scriptless**
- ❑ B. tag1 body-content is **JSP**
  tag2 body-content is **empty**
  tag3 body-content is **scriptless**
- ❑ C. tag1 body-content is **JSP**
  tag2 body-content is **JSP**
  tag3 body-content is **JSP**
- ❑ D. tag1 body-content is **scriptless**
  tag2 body-content is **JSP**
  tag3 body-content is **JSP**
- ❑ E. tag1 body-content is **JSP**
  tag2 body-content is **scriptless**
  tag3 body-content is **scriptless**

6 Assuming the appropriate **taglib** directives, which are valid examples of custom tag usage? (Choose all that apply.)

- ❑ A. **<foo:bar />**
- ❑ B. **<my:tag></my:tag>**
- ❑ C. **<mytag value="x" />**
- ❑ D. **<c:out value="x" />**
- ❑ E. **<jsp:setProperty name="a" property="b" value="c" />**

7 Given the following scriptlet code:

```
11. <select name='styleId'>
12. <% BeerStyle[] styles = beerService.getStyles();
13.    for ( int i=0; i < styles.length; i++ ) {
14.      BeerStyle style = styles[i]; %>
15.   <option value='<%= style.getObjectID() %>'>
16.     <%= style.getTitle() %>
17.   </option>
18. <% } %>
19. </select>
```

Which JSTL code snippet produces the same result?

- ❑ A.
```
<select name='styleId'>
          <c:for array='${beerService.styles}'>
            <option value='${item.objectID}'>${item.title}</option>
          </c:for>
        </select>
```
- ❑ B.
```
<select name='styleId'>
  <c:forEach var='style' items='${beerService.styles}'>
    <option value='${style.objectID}'>${style.title}</option>
  </c:forEach>
</select>
```
- ❑ C.
```
<select name='styleId'>
  <c:for var='style' array='${beerService.styles}'>
    <option value='${style.objectID}'>${style.title}</option>
  </c:for>
</select>
```
- ❑ D.
```
<select name='styleId'>
  <c:forEach var='style' array='${beerService.styles}'>
    <option value='${style.objectID}'>${style.title}</option>
  </c:for>
</select>
```

Chapter 9 Answers

1 Which is true about TLD files?

(JSP v2.0 pgs 3-16, 1-160)

☑ A. TLD files may be placed in any subdirectory of **WEB-INF**.

☐ B. TLD files are used to configure JSP environment attributes, such as **scripting-invalid**.

-Option B is invalid because TLD files configure tag handlers not the JSP environment.

☐ C. TLD files may be placed in the **META-INF** directory of the WAR file.

-Option C is invalid because TLD files are not recognized in the META-INF of the WAR file.

☐ D. TLD files can declare both Simple and Classic tags, but TLD files are NOT used to declare Tag Files.

-Option D is invalid because Tag Files may be declared in a TLD (but it is rare).

2 Assuming the standard JSTL prefix conventions are used, which JSTL tags would you use to iterate over a collection of objects? (Choose all that apply.)

(JSTL v1.1 pg. 42)

☐ A. **<x:forEach>**

-Option A is incorrect as this is the tag used for iterating over XPath expressions.

☐ B. **<c:iterate>**

-Option B is incorrect because no such tag exists.

☑ C. **<c:forEach>**

☐ D. **<c:forTokens>**

-Option D is incorrect because this tag is used for iterating over tokens within a single string.

☐ E. **<logic:iterate>**

☐ F. **<logic:forEach>**

-Options E and F are incorrect because the prefix 'logic' is not a standard JSTL prefix (this prefix is typically used by tags in the Jakarta Struts package).

3 A JSP page contains a **taglib** directive whose **uri** attribute has the value **myTags**. Which deployment descriptor element defines the associated TLD?

(JSP v2.0 pgs 3-12,13)

- ❑ A.
```
<taglib>
  <uri>myTags</uri>
  <location>/WEB-INF/myTags.tld</location>
</taglib>
```
- ❑ B.
```
<taglib>
  <uri>myTags</uri>
  <tld-location>/WEB-INF/myTags.tld</tld-location>
</taglib>
```
- ❑ C.
```
<taglib>
  <tld-uri>myTags</tld-uri>
  <tld-location>/WEB-INF/myTags.tld</tld-location>
</taglib>
```
- ☑ D.
```
<taglib>
  <taglib-uri>myTags</taglib-uri>
  <taglib-location>/WEB-INF/myTags.tld</taglib-location>
</taglib>
```

– Option D specifies valid tag elements.

4 A JavaBean **Person** has a property called **address**. The value of this property is another JavaBean **Address** with the following string properties: **street1**, **street2**, **city**, **stateCode** and **zipCode**. A controller servlet creates a session-scoped attribute called **customer** that is an instance of the **Person** bean.

(JSTL v1.1 pg 4-28)

Which JSP code structures will set the **city** property of the **customer** attribute to the **city** request parameter? (Choose all that apply.)

- ❑ A.
```
${sessionScope.customer.address.city = param.city}
```
- ☑ B.
```
<c:set target="${sessionScope.customer.address}"
       property="city" value="${param.city}" />
```
- ❑ C.
```
<c:set scope="session" var="${customer.address}"
       property="city" value="${param.city}" />
```
- ☑ D.
```
<c:set target="${sessionScope.customer.address}"
       property="city">
  ${param.city}
</c:set>
```

–Option A is invalid because EL does not permit assignment.

–Option C is invalid because the var attribute does not accept a runtime value, nor does it work with the property attribute.

5 Which **<body-content>** element combinations in the TLD are valid for the following JSP snippet? (Choose all that apply.)

(JSP v2.0 Appendix JSP.C specifically pgs 3-21 and 3-30)

```
11. <my:tag1>
12.    <my:tag2 a="47" />
13.    <% a = 420; %>
14.    <my:tag3>
15.      value = ${a}
16.    </my:tag3>
17. </my:tag1>
```

-Tag1 includes scripting code so it must have at least 'JSP' body-content. Tag2 is only shown as an empty tag, but it could also contain 'JSP' or 'scriptless' body-content. Tag3 contains no scripting code so it may have either 'JSP' or 'scriptless' body-content.

- ☐ A. tag1 body-content is **empty**
  tag2 body-content is **JSP**
  tag3 body-content is **scriptless**
- ☑ B. tag1 body-content is **JSP**
  tag2 body-content is **empty**
  tag3 body-content is **scriptless**
- ☑ C. tag1 body-content is **JSP**
  tag2 body-content is **JSP**
  tag3 body-content is **JSP**
- ☐ D. tag1 body-content is **scriptless**
  tag2 body-content is **JSP**
  tag3 body-content is **JSP**
- ☑ E. tag1 body-content is **JSP**
  tag2 body-content is **scriptless**
  tag3 body-content is **scriptless**

-Option A is invalid because tag1 cannot be 'empty'.

-Option D is invalid because tag1 cannot be 'scriptless'.

6 Assuming the appropriate **taglib** directives, which are valid examples of custom tag usage? (Choose all that apply.)

(JSP v2.0 section 7)

- ☑ A. **<foo:bar />**
- ☑ B. **<my:tag></my:tag>**
- ☐ C. **<mytag value="x" />**
- ☑ D. **<c:out value="x" />**
- ☐ E. **<jsp:setProperty name="a" property="b" value="c" />**

-Option C is invalid because there is no prefix.

-Option E is invalid because this is an example of a JSP standard action, not a custom tag.

7 Given the following scriptlet code:

(JSTL v1.1 pg 6-48)

```
11. <select name='styleId'>
12. <% BeerStyle[] styles = beerService.getStyles();
13.    for ( int i=0; i < styles.length; i++ ) {
14.      BeerStyle style = styles[i]; %>
15.   <option value='<%= style.getObjectID() %>'>
16.     <%= style.getTitle() %>
17.   </option>
18. <% } %>
19. </select>
```

Which JSTL code snippet produces the same result?

- ❑ A.
```
<select name='styleId'>
        <c:for array='${beerService.styles}'>
          <option value='${item.objectID}'>${item.title}</option>
        </c:for>
      </select>
```
- ☑ B.
```
<select name='styleId'>
  <c:forEach var='style' items='${beerService.styles}'>
    <option value='${style.objectID}'>${style.title}</option>
  </c:forEach>
</select>
```
- ❑ C.
```
<select name='styleId'>
  <c:for var='style' array='${beerService.styles}'>
    <option value='${style.objectID}'>${style.title}</option>
  </c:for>
</select>
```
- ❑ D.
```
<select name='styleId'>
  <c:forEach var='style' array='${beerService.styles}'>
    <option value='${style.objectID}'>${style.title}</option>
  </c:for>
</select>
```

-Option B is correct because it uses the proper JSTL tag/attribute names.