Introduction to Java Server Pages

Introduction

- Java Server Pages allow special tags and Java code to be embedded in HTML files. These tags and code are processed by the Web server to dynamically produce a standard HTML page for the browser.
  - Another architecture in the Web-based distributed application arsenal.
  - Produce dynamic Web pages on the server side (like Servlets), but separate application logic from the appearance of the page.
  - The tags allow previously compiled Java code, in the form of JavaBeans, to be used.
  - Allows fast development and testing.
  - May also produce XML documents, instead of HTML.

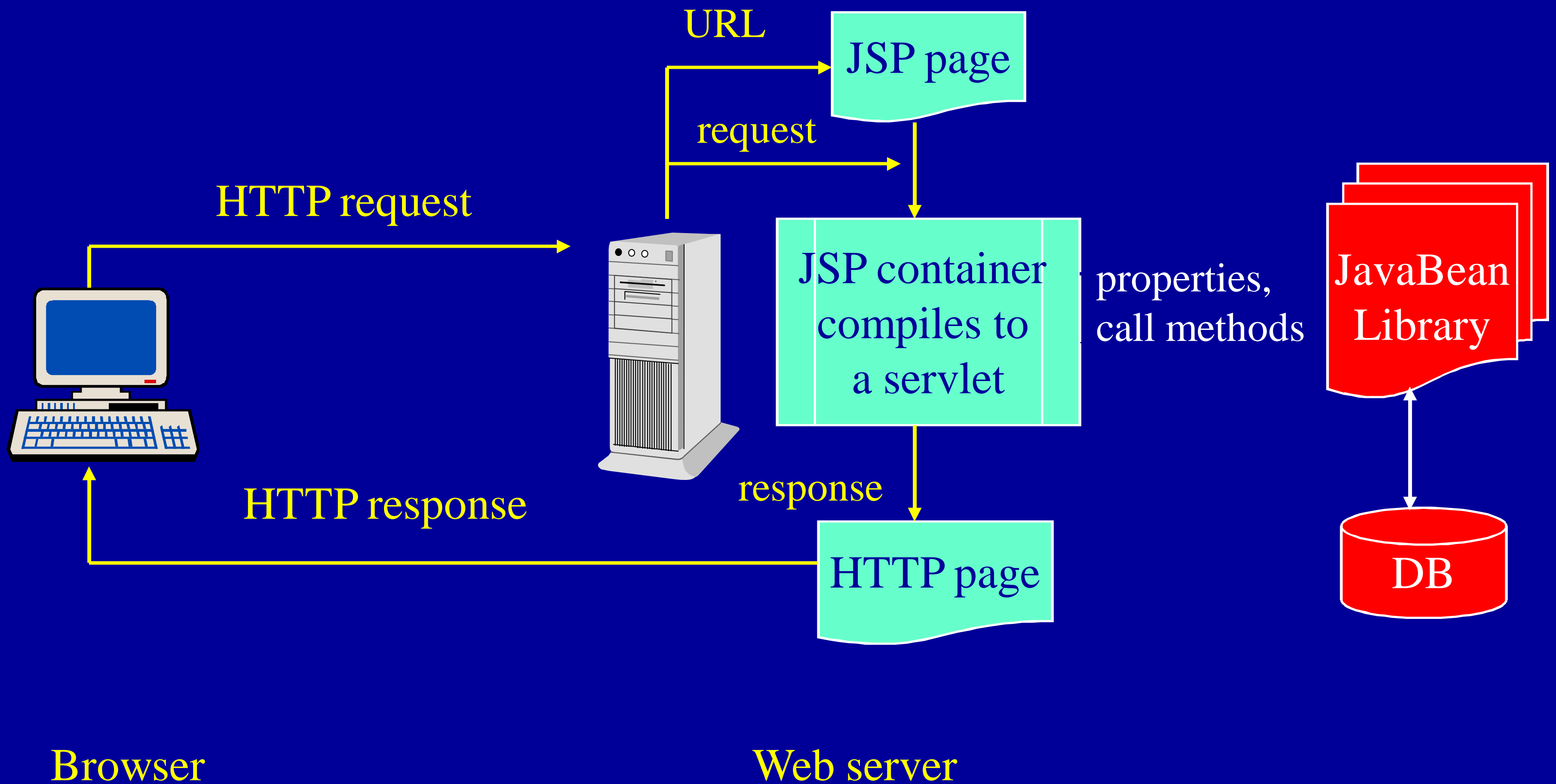
Development of JSP

- Java Server Pages were developed as a response to Microsoft's Active Server Pages (ASP). The main differences are that ASP only runs on Microsoft IIS and Personal Web Servers, and JSP has user-defined tags.
- Development dates: (Note that JSP is built on top of Servlets)
  - Servlet 2.1 Jan. 99
  - JSP 1.0 June 99
  - Source code released to Apache to develop Tomcat server November 99
  - Servlet 2.2 and JSP 1.1 (J2EE1.2) December 99
  - Look for further development of tag library in 00.

JSP elements

- A JSP page looks like a standard HTML or XML page with additional elements processed by the JSP container.
- Typically, these elements create text that is inserted into the resulting document.
- JSP elements include:
  - Scriptlet enclosed in **<%** and **%>** markers: a small script in Java to perform arbitrary functions. Executed in the underlying servlet context.
  - Expression: anything between **<%=** and **%>** markers is evaluated by the JSP engine as a Java expression in the servlet context.
  - JSP directive enclosed in **<%@** and **%>** markers—passes information to the JSP engine (guides “compilation”).
  - JSP actions or tags are a set of customizable XML-style tags for particular actions, e.g. predefine **jsp:useBean** instantiates a Java Bean class on the server.

Architecture



“Hello User” Servlet, Revisited

```
import java.io.* ;
import javax.servlet.* ;
import javax.servlet.http.* ;
public class HelloWorld extends HttpServlet {
    public void doGet(HttpServletRequest request,
                              HttpServletResponse response)
                throws IOException, ServletException {
        response.setContentType(“text/html”) ;
        PrintWriter out = response.getWriter() ;
        out.println(“<html><body>”) ;
        out.println(“<h1>Hello ” +
                    request.getParameter(“who”) + “!</h1>”) ;
        out.println(“</html></body>”) ;
    }
}
```

An Equivalent JSP Page

```
<html> <body>
<h1>Hello <%= request.getParameter("who") %> </h1>
</body> </html>
```

Remarks

- This text may be saved in a file **hellouser.jsp** in the **dbc/** application directory.
- A suitable form element to front-end this page might be:

```
<form action="hellouser.jsp">
        Name:  <input type=text name=who size=32> <p>
        <input type=submit>
</form>
```

- The JSP version is *much* more compact and easier to understand!
- We can expect this to be generally the case when the logic behind a dynamic page is relatively simple, and the bulk of the content is static HTML.

Overview of JSP

What is a JSP Page?

- According to the JavaServer Pages™ Specification:
  - "A JSP page is a text-based document that describes how to process a *request* to create a *response*. The description intermixes template data with some dynamic actions [. . .]."
- JSP deliberately supports multiple paradigms for authoring dynamic content.
  - Notably, scriptlets, JavaBeans and Tag Libraries.

Translating and Executing JSP Pages

- A JSP page is executed in a JSP container, generally installed in a Web server.
  - Think of a "JSP container" as a JVM with suitable software installed.
- The underlying semantic model is that of a servlet.
- A typical JSP container will translate the JSP page to a Java servlet.
- By default translation and compilation of a JSP page is likely to occur the first time it is accessed.
  - With Tomcat 3.1, you can find the generated Java and the class files in a subdirectory under **jakarta-tomcat/work**.

JSP Features

- **Standard directives** guiding translation of a JSP page to a servlet.
- **Standard actions** in the form of predefined JSP tags.
- **Script language *declarations*, *scriptlets*, and *expressions*** for including Java (or in principle other language) fragments that embed dynamic content.
- **A portable tag extension mechanism**, for building tag libraries—effectively extending the JSP language.

Fixed Template Data

- These are the parts of a JSP page that are used verbatim in the response.
  - Or in input to JSP actions—see later.
- In simple cases this data will take the form of plain HTML.
- This text can be written anywhere in the JSP file. It is copied unchanged, in lexical order, to the output stream of a response

Directives vs Actions

- The special JSP elements include directives and actions.
- *Directives* provide "global" information about the behavior of a JSP page—independent of any specific request.
  - In practice, they guide the process of translation from JSP to Java.
- An *action* has some effect in the context of a particular request.
  - e.g. generating some output that depends on form parameters, or creating an object in the JSP container.
- In other words, an action is a "request-time" operation.
- This boils down to the more familiar distinction between compile-time and run-time commands.

Actions vs Scripting Elements

- Scripting elements (the other kind of element) have a similar status to action elements—they are "request-time" operations.
- They are distinguished by syntax:
  - Actions follow a strict XML syntax—scripting elements are largely unconstrained Java "inserts".
- They are also distinguished at a software engineering level:
  - Scripting elements provide a quick way to achieve an effect, but depend on knowledge of the underlying servlet model.
  - Actions and the associated tag libraries provide a potentially higher level of abstraction (meta-linguistic abstraction?)
- As with all good libraries, actions should be easy to use, but they are relatively hard work to implement.

JSP Directives

- **page** directives communicate page-specific information to the JSP engine, such as buffer or thread information or specify an error page.
- **include** directives includes an external document in the page. These are often company information files or copyright files, but can also be jsp files.
- **taglib** directive indicates a library of custom tags that the page can include.
- **language** directives specify script language + possible extensions. The only standard language is Java.
- All are delimited by **<%@** and **%>** markers, e.g.:

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

Standard Actions

- Predefined action tags include:
  - jsp:useBean - declares the usage of an instance of a JavaBeans component. If it does not already exist, then the instance is created and registered in the servlet environment.
  - jsp:setProperty - this tag can be used to set all the properties of a Bean from the request parameter stream with parameters of the same name, or can be used to set individual properties.
  - jsp:getProperty - gets the property of the Bean, converts it to a String, and puts it into the implicit object “out”.
  - jsp:forward - forwards the request to another jsp page or servlet.
  - jsp:include - include another JSP page
  - jsp:plugin - load into the specified plug-in of the browser
- All have a pure XML syntax, e.g.:

<jsp:useBean id=“clock” class=“calendar.jspCalendar” />

Scripting Elements

- There are three kinds of scripting element:
  - **Scriptlets**: arbitrary fragments of Java that are executed in the request handling method. Syntax is e.g.:

    ```
    <%  out.println(“Your number is ” + Math.random()) ;  %>
    ```

  - **Expressions**: any Java expression, cast to a **String**. Evaluated in the context of the request handling method. Syntax is e.g.:

    ```
    <%= Math.sqrt(2) %>
    ```

  - **Declarations**: any Java declaration that can appear at the class body level, such as an instance variable or method declaration. Syntax is e.g.:

    ```
    <%! Color c = new Color (0, 128, 255) ; %>
    ```

Scripting Elements

The Role of Scripting Elements

- As explained in the previous lecture, scripting elements represent one of three basic kinds of element in JSP.
  - The other two are directives and (standard and customizable) actions.
- While only one part of the larger picture—and perhaps lacking the refinement that can be achieved with customized action elements—scripting elements by themselves allow one to do essentially all the interesting things that could be done with servlets.
- Writing a JSP page with scripting elements may be much less work than writing a servlet. The source is shorter—it doesn't need all the boilerplate code—and compilation is handled automatically.
  - In simple cases deploying the JSP page may be no more trouble than deploying a static HTML page.

Is this Related to JavaScript?

- **No!** There is no relation between the scripting elements in JavaServer Pages and Netscape's somewhat older JavaScript language.

Scriptlets

- The most basic kind of scripting element is the *scriptlet*.
- A scriptlet is an arbitrary fragment of Java code.
- This Java code is just copied to the request handling method in the servlet code, generated when the JSP page is translated.
- The placement of the scriplet code in the generated servlet just reflects the position of the scriplet element in the JSP source.
  - Reasonably enough, it follows the generated code that outputs the preceding template text, and precedes the generated code that outputs the following template text.
- The syntax of a scriplet element is:

<% Java code in here %>

A JSP Page with a Scriplet

```
<html><head></head><body>
<%  out.println("Now is" + new java.util.Date()) ;  %>
</body></html>
```

- This is the complete JSP page. We just save this text in, e.g., "**date.jsp**", and install it in a suitable document directory.
- Don't forget the semicolon!
  - We will see shortly that a scriplet does not always have to be a self-contained Java statement, but it must yield a legal Java program when it is inserted in the generated servlet code.
  - Mistakes here may lead to odd compiler messages, relating to the surrounding, automatically generated parts of the servlet code!
  - Since compilation is done on the fly, such messages appear the first time you visit the page.

Viewing the **date.jsp** Page

- If we visit, e.g. **http://sirah:8081/dbc/date.jsp** , we see something like:

Now is Wed Nov 29 10:28:51 EST 2000

Predefined Variables

- The example referenced a variable **out**.
- This was not declared. As one may guess it stands for the output stream associated with the response.
- **out** is one of several predefined variables that can be accessed in JSP scripting elements.
- Others include **request**, **response**, **session**, etc.

Meanings of Predefined Variables

- **request**
  The **HttpServletRequest** object passed to the servlet's request handling method.
- **response**
  The **HttpServletResponse** object passed to the servlet's request handling method.
- **out**
  The **PrintWriter** object associated with **response**.
- **session**
  The **HttpSession** object associated with the request.

More Predefined Variables

- **application**
  An object of type **ServletContext** object associated with the application. Like **HTTPSession**, this can cache values. Unlike **HTTPSession**, the stored values are shared by all servlet invocations in the current "application" (which typically corresponds to the servlet context).
- **config**
  The **ServetConfig** object associated with this page (see, for example, the discussion of initialization parameters in the lectures on servlets).
- **pageContext**
  An object of type **PageContext** . Something like **ServletContext**, but values stored here are only shared by invocations of this page. Thus very similar in behavior to servlet instance variables.

"Compound" Scriptlets

- A compound statement in the generated code does not have to be limited to a single scriptlet element.
- Instead can be assembled from several scriptlet elements, surrounding blocks of template text.
- Here is an example using a conditional around blocks of template data (taken from the book by Hall):

```
<html><head></head><body>
<%  if (Math.random() < 0.5) {  %>
Have a nice day!
<% } else { %>
Have a lousy day!
<% } %>
</body></html>
```

Expressions

- A JSP *expression* is very similar to a scriplet, except that the enclosing Java code is required to be an expression.
- The expression is evaluated, then converted to a **String**.
- The string is effectively interpolated into the template text—wherever the element appears—and sent to the browser as part of the generated HTML.
- The syntax of a expression element is:

  <%= Java expression in here %>

  – Distinguished from a scriptlet by the **=** sign attached to **<%** .

Avoiding **out.println()**

- We can use an expression to replace the first scriplet in this lecture, viz:

```
<html><head></head><body>
<%  out.println("Now is" + new java.util.Date()) ;  %>
</body></html>
```

with:

```
<html><head></head><body>
Now is <%= new java.util.Date()  %>
</body></html>
```

- In general, expression elements save us writing many scriplets that just include calls to **out.println()**.

Loops and Tables

- Strictly speaking the next example does not introduce any new features, but it illustrates how to combine some of the elements we already know.
- It is based on one of our "form" examples. Recall this form presented a **SELECT** element for choosing several pets from a list.
- This JSP page replaces the servlet **MultiValue**.

Showing Pets

```
<html><head></head><body>
Your pets:<p>
<table border cellspacing=0 cellpadding=5>
<%
String [] pets = request.getParameterValues("pets") ;
for (int i = 0 ; i < pets.length ; i++) {
%>
    <tr><td>  <%= pets [i] %>  </td></tr>
<%
}
%>
</table>
</body></html>
```

Declarations

- The last kind of JSP scripting element is the *declaration*.
- In contrast to expressions, JSP declarations give us a fundamentally new functionality, which cannot be achieved with scriptlets alone.
- They allow code to be inserted at the top level of the Servlet class definition, outside the the body of the request-handling method.
- The syntax of a declaration element is:

  <%! Java declaration in here %>

  – Distinguished from a scriptlet by the **!** sign attached to **<%** .

The Counter Servlet Revisited

```
import java.io.* ;
import javax.servlet.* ;
import javax.servlet.http.* ;
public class Counter extends HttpServlet {
    int count = 0 ;
    public void doGet(HttpServletRequest req,
                                    HttpServletResponse resp)
                      throws IOException, ServletException {
        resp.setContentType(“text/html”) ;
        PrintWriter out = resp.getWriter() ;
        out.println(“<html><head></head><body>”) ;
        out.println(“This servlet instance has been accessed ” +
                         (count++) + “ times”) ;
        out.println(“ </body></html>”) ;
    }
}
```

A JSP Counter

```
<%!  int count = 0 ;  %>
<html><head></head><body>
This JSP page has been accessed <%=  count++  %> times
</body></html>
```

- As usual, this is the *complete* JSP page.
- Note that actually it doesn’t matter where the declaration element appears in the JSP page, since it is not associated with request handling “thread of control”.

Method Declarations

- A JSP declaration element can include a method declaration, e.g.:

```
<%!
     String  myMethod() {
          return "madness" ;
     }
%>
<html><head></head><body>
There is <%=  myMethod()  %> in my method.
</body></html>
```

- For page initialization, use a JSP declaration to override:

```
public void jspInit() {}
```

Local Variables

- Use a JSP declaration if you want to declare an instance variable of a servlet.
- Do not use a JSP declaration if you just want a declare a variable local to the request handling method.
- Local variables should be declared in ordinary scriplet elements, e.g.:

```
<%  int loopCount ;  %>
```

Directives

- Strictly speaking, JSP directives are separate from scripting elements, but it is convenient to review them here.
- The general syntax is:

  <%@ name attr1="value1" attr2="value2" . . . %>

  where ***name*** is the name of the directive, and ***attr1***, ***attr2***, . . . are the names of various attributes associated with this directive.
- The currently allowed values for name are **page, include** and **taglib**:
  - The **page** directive defines various attributes associated with a page.
  - The **include** directive allows another file to be textually included.
  - The **taglib** directive imports a tag library.

The **import** attribute

- For now the most important attribute that can be set by the **page** directive is the **import** attribute.
- This simply adds an import directive to the generated Java code.
- Syntax is, e.g.:

```
<%@   page  import="java.util.*"  %>
```

The Vending Machine Revisited

- We will convert the final version of the Vending Machine servlet application (two servlets) to JSP.
- This is interesting as a case where there is more “logic” than template text.
- Nevertheless the JSP version is somewhat shorter, and arguably more readable.

Vending Machine (Preamble)

```
<%@  page import="java.util.*" %>

<%!
String [] snacks = {"Chips", "Popcorn", "Peanuts", . . . } ;
%>

<%
Vector selections = (Vector) session.getAttribute("selections")
    ;
if (selections == null) {
    selections = new Vector() ;
    session.setAttribute("selections", selections) ;
}
String selection = request.getParameter("selection") ;
if (selection != null)
    selections.addElement(selection) ;
%>
```

Vending Machine (Main Body)

```
<html><head></head><body>
<%
for (int = 0 ; i < snacks.length ; i++)
%>
      <form action=vendingmachine.jsp>
      <input type=submit  name=selection  value=<% snacks [i] %>
    >
       </form>
<%
}
%>
<a href=vendingview.jsp>View current selections</a>
</body></html>
```

The Selection Viewing Page

```
<%@  page import="java.util.*" %>
<html><head></head><body>
Current selections:<p>
<%
Vector selections = (Vector) session.getAttribute("selections")
    ;
if(selections != null)
    for (int j = 0 ; j < selections.size() ; j++)
        out.println(selections.get(j) + "<br>") ;
%>
<a href=vendingmachine.jsp>Select more</a>
</body></html>
```

Actions and Beans

Features of Scripting Elements

- We saw in the last lecture that the scripting elements of JSP are very powerful.
- They give us essentially all the power of Java and servlets, but with the code embedded directly in an HTML page.
- Code fragments and HTML are simply separated by **<%**, **%>** brackets.
- This is both an advantage and a disadvantage:
  - It allows a dynamic page to be developed and deployed very quickly.
  - Equally quickly, the JSP source can become difficult to read.
- The same remarks may presumably be applied to syntactically similar systems (e.g. PHP).

Disadvantages of Scriptlets

- An HTML page containing many Java inserts can be difficult to read and maintain.
- This may be a particular problem when large Web application development teams are involved—typically divided into Web site designers and programmers.
- The former may be happy with HTML, and the latter may be happy with Java. Merging both languages in the same file potentially forces *both* teams to be involved in maintaining the file.
- This is a poor separation of concerns.

Actions

- JSP *actions* are special elements that follow a strict XML syntax.
- They sit naturally within an HTML document (and presumably are comfortable to Web designers).
- If actions are used exclusively, there need not be any Java source embedded in the HTML page.
- However we can still invoke Java code through *JavaBeans*.
- The actions we are concerned with in this lecture are the predefined *standard actions*.

The Standard Actions

- **jsp:forward**
  Forwards the request to another JSP page or servlet.
- **jsp:include**
  Include another JSP page
- **jsp:plugin**
  Include an applet in the page.
- **jsp:useBean**
  Declares usage of a JavaBeans component in the page.
- **jsp:setProperty**
  Sets a property of a bean.
- **jsp:getProperty**
  Gets a property of the Bean.

Syntax

- All actions have a strict XML syntax.
- An element either has the form:

  < name attr1="value1" attr2="value2" . . . />

  or the form:

  < name attr1="value1" attr2="value2" . . . >
  JSP body of element
  </ name >

- Here ***name*** is the name of the element, and ***attr1***, ***attr2***, . . . are the names of various ***attributes*** associated with this element.
- Differences from HTML:
  - Case is significant.
  - Attribute values must always be quoted.
  - Single-tag elements (elements with no body) are closed by ***/>***.

The Forward Action

- The **jsp:forward** action transfers control from the current JSP page to another location on the server.
- Some possible uses:
  - Send the contents of a display page at the end of a page that mainly does some actions.
  - In a conditional scriptlet, forward control to some error page when a particular situation arises.
- In some of the earlier servlet examples we used **sendRedirect()** to transfer to a display page after doing some actions.
- Using **jsp:forward** would be more efficient, because it does not involve the browser in the transfer.

Example

- We define the file **forward:jsp**:

```
<html><head></head> <body>
<%
String who = request.getParameter(“who”) ;
if (who == null) {
%>
    <jsp:forward  page=“noparam.html” />
<%
} else {
%>
    <h1>Hello <%=  “who” %> </h1>
<%
}
%>
</body> </html>
```

Remarks

- If the **who** parameter is not defined, control is forwarded to the **noparam.html**.
- Note the output buffer is cleared before transferring control.
- Output already generated by the current page is lost.
  - The fragment

    <html><head></head> <body>

    from this page is not delivered to the browser.
- Incidentally, this example illustrates the case that JSP with heavy use of scriplets can be relatively hard to read!

The **jsp:param** Element

- Some of the standard actions allow **jsp:param** elements in their body. **jsp:forward** is a case in hand.
- We could replace our static “missing parameter” error page, **noparam.html**, with a JSP page that itself takes a parameter, specifying which parameter was missing from the original request.
- The **jsp:forward** element would be replaced with:

```
<jsp:forward page="noparam.jsp">
     <jsp:param  name="missing"  value="who" />
</jsp:forward>
```

The Include Action

- The **jsp:include** action temporarily transfers control from the current JSP page to another location, and includes the text generated by the other page in its own output.
- Syntax is, e.g.:

```
<jsp:include  page="banner.html" />
```

- As with **jsp:forward**, **jsp:param** elements may be optionally included in a body.
- This is subtly different from the **include** *directive*, which includes text, *before* translation of the JSP.

The Plug-in Action

- The **jsp:plugin** tag introduces an element that allows an applet to be included the generated page, in such a way that it can be displayed by the Java plug-in in a browser.
- This is a replacement for running the HTML conversion program over a file with an **applet** tag.
- Example (from the JSP specification):

```
<jsp:plugin type=applet  code="Molecule.class"
                         codebase="/html">
    <jsp:params>
        <jsp:param name="molecule"
                   value="molecules/benzene.mol"/>
    </jsp:params>
    <jsp:fallback>
        <p> unable to start plugin </p>
    </jsp:fallback>
</jsp:plugin>
```

JavaBeans

- The other standard actions relate to manipulation of JavaBeans within a JSP page.
- JavaBeans is a component architecture.
- It dictates a set of rules that software developers must follow to create reusable components.
- A Bean is an instance of a class that was coded following these rules.
- JSP interacts with Beans through tags (naturally).

Properties

- Any Bean has a set of named *properties*.
- A property is something like an instance variable.
  - Whether it corresponds to an actual instance variable of the object is an internal implementation choice.
- In any case properties can be read or written using “getter” and “setter” methods.
- If the property has name ***name*** and type ***Type*** these methods have the form:

  public *Type* get*Name*()

  public void set*Name*(*Type* value)

  - The property name starts with a lower case letter, which is capitalized in in the name of the **get**/**set** method.
  - Some properties may we read-only or write-only, without associate **set** or **get** methods.

Trigger Properties

- A Bean can have additional methods besides **get** and **set** methods.
- One advantage of only using the standardized methods is that various sorts of *Bean Container* application can invoke them automatically.
- Writing to, or reading from a *trigger property* will cause the Bean to perform some non-trivial activity “behind the scenes”.
  - They could cause an external event, or just some internal computation.
- If accessing one property causes the values of other properties to change, these are called *linked properties*.

Constructor

- A JavaBean must also have a no-argument constructor. This allows a Bean container to instantiate the class without prior knowledge of the class.
  - By using **Class.newObject()**

Elementary Use of Beans in JSP

- The **jsp:useBean** action tells a page that a particular Bean is to be used here.
- Its minimal form names a Bean, and specifies its class.
- If there is no Bean of the specified name currently “in scope” (the meaning of this will be discussed later) the **jsp:useBean** method creates a new Bean instance.
- The action can also include text to initialize the bean, if it is freshly created.

The “Use Bean” Element.

- The two basic forms of the element are:

  <jsp:useBean id=“*bean-name*” class=“*class-name*” />

  or

  <jsp:useBean id=“*bean-name*” class=“*class-name*”>
  conditional “initalization” text
  </jsp:useBean>

- The name ***bean-name*** should be a legal Java variable name
- Translation of the JSP page causes a variable of this name, and type ***class-name***, to be declared (in the equivalent servlet code).
- In the second form, the “conditional text” is only processed if the Bean is freshly created.

Getting and Setting Bean Properties

- The simplest form of the **jsp:getProperty** and **jsp:setProperty** elements are:

  <jsp:getProperty name="*bean-name*" property="*property*">

  <jsp:setProperty name="*bean-name*"
  property="*property*" value="*value*">

- Here ***name*** is the Bean name specified in the **jsp:useBean** tag; ***property*** is the name of a property in its class.
- The term ***value*** can be a literal string. The JSP container will attempt to convert it to the type of the argument of the associated **set** method.
  - This will only work if the argument is **String** or a Java primitive type.
- The term ***value*** can also be a JSP expression element.
  - Note this represents an unusual nesting of elements.

A Simple Example

- We define the Java Bean class:

```
import java.util.* ;
class DateBean {
    public String getDate() {
        return  (new Date()).toString()  ;
    }
}
```

- This has a single read-only property called **date** (note there is no associated instance variable).
- It implicitly has a no-argument constructor: the default constructor.
- Under Tomcat this class can be placed under the application **WEB-INF/classes/** directory.

JSP Page Using the Date Bean

```
<%@  page  import="date.*"  %>

<jsp:useBean  id="now"  class="DateBean"/>

<html><head></head><body>
Now is <jsp:getProperty  name="now"  property="date"/>
</body></html>
```

Initializing Beans from the Request

- If the **value** attribute is omitted from a **jsp:setProperty** element, the JSP container looks for a (form) parameter with the same name as the property, and sets the property using the value of that parameter.
- For example, if the a JavaBean class **UserBean** has a property **userName**, this property could be initialized as follows:

```
<jsp:useBean  id="user"  class="UserBean">
     <jsp:setProperty  name="user" property="userName">
</jsp:useBean>
```

The value of the **userName** property will be taken from the value of a form parameter called **userName**.

Controlling a Bean's Scope

- The accessibility and lifetime of a Bean created by a JSP page is controlled by the **scope** attribute in the **jsp:useBean** element.
- The possible values are

  page
  Current page only. Like a local variable in a request-handling method.

  request
  Similar to **page**, but passed to other pages if control is forwarded from this page.

  session
  The current session. Like an attribute of the **HttpSession** object.

  application
  Available to any subsequent request in the same Web application (servlet context).

Session Beans

- The most interesting case is probably **session** scope.
- By declaring beans with session scope we can avoid having to deal explicitly with the **HttpSession** object.
- For example, I could create a Bean class that absorbed the functionality of the **DBSession** class from the student database example in the preceeding lecture set.
- This might be initialized as follows:

```
<jsp:useBean  id="dbs"  class="DBSessionBean"
                   scope="session">

      <jsp:setProperty  name="dbs" property="userName">
      <jsp:setProperty  name="dbs" property="password">
      <jsp:setProperty  name="dbs"
                         property="connected"  value="true">
</jsp:useBean>
```

An Example

Students Database Revisited

- We will recast the servlet+JDBC implementation of a student database into a JSP+JDBC version.
- This illustrates practical use of a session bean, and will shed some light on the advantages and limitations of JSP standard actions plus JavaBeans.

A Database Session Bean

- We will use one Java bean.
- This will be a more evolved version of the **DBSession** class in the earlier servlet implementation.
- In our earlier implementation, the **DBSession** class was responsible only for opening and closing the connection with the database.
- The **DBSession** class exposed a JDBC **Statement** object to the servlet code.
- The servlet code itself was responsible for executing SQL statements through the mediation of this object.

Moving Logic into the Bean

- A motivating goal of the JSP actions+Beans paradigm is to obtain a clean separation between presentational aspects and computational logic aspects.
- As much of the logic as possible should be isolated in the Bean, and as much of the HTML generation as possible should be isolated in the JSP page.
  - We will see that the standard actions allow us to make a reasonable job of this, although not perfect.
- An important change, therefore, will be to shift generation of SQL queries from the servlet (now JSP) layer, to the Bean layer.

Session Tracking

- In the new paradigm, session tracking can be even simpler than with servlets.
- In the servlet implementation, recall, we needed code like:

```
HttpSession session = request.getSession(true) ;
dbs = (DBSession) session.getAttribute("dbs") ;
if(dbs == null) {                    // Session new or timed out
   dbs = new DBSession() ;
   session.setAttribute("dbs",  dbs) ;
   session.setMaxInactiveInterval(300) ;    // 5 minutes.
}
```

at the beginning of servlets that used session state (in this case the object **dbs**).

Session Tracking Code in JSP

- An equivalent action in JSP:

```
<jsp:useBean  id="dbs"  class="DBSessionBean"
   scope="session">
     <%  session.setMaxInactiveInterval(300) ; %>
</jsp:useBean>
```

- Recall the **jsp:useBean** element first looks for an *existing* Bean of the specified name in the specified scope.
- With **session** scope, it implicitly looks in the **HttpSession** object for an attribute of this name.
- If the action doesn't find an existing Bean, it creates a new one using the standard, no-argument Bean constructor.
- The JSP statements in the *body* of the **jsp:useBean** element are evaluated only if a new Bean was created.
- Thus this JSP element exactly reproduces the servlet code.

The Selection Page

- Apart from establishing a new session if necessary, the selection page of the application is responsible for finding all key values from the database table, and printing them in an HTML **select** element (in a form).
- Clearly the database query should go in the JavaBean code, and clearly the general frame of the form should go in the JSP.
- There is a problem with the generation of the **select** element.
  - If it goes entirely in the JSP, we have to resort to general scripting elements for iteration.
  - If it goes in the JavaBean, then the JavaBean has to generate at least some HTML, for the rows of the **select** element.
- Either way, there is some conflict with the goal of separating presentation from logic.

Selection Form Generation

- The lesser evil here seems to be including a little HTML generation code in the Bean. The form-generating JSP for the selection page is:

```
<html><head></head><body>
<form  action=view.jsp>
    Select by ID:<br>
    <jsp:getProperty name="dbs"  property="menu"/>
    View selected record: <input  type=submit  value="View">
</form>
</body></html>
```

- The property **menu** of the Bean is assumed to be a string containing a suitable HTML **select** element.

The **DBSessionBean** Class

- The **DBSessionBean** class starts off in a very similar way to the old **DBSession** class.
- Its constructor opens a connection to a suitable database.
- It will implement **HttpSessionBindingListener**, and the **valueUnbound()** event-dispatching method is responsible for closing the connection.
- The new class has a read-only property called **menu**, which returns a suitable **select** element. This is defined by the following method on the Bean class. . .

Computing the **menu** Property

```
public String getMenu() throws SQLException {
    // Extract keys from table
    ResultSet rs = stat.executeQuery(“SELECT login FROM  ” +
   table) ;
    // Generate HTML select element
    StringBuffer menu = new StringBuffer() ;
    menu.append(“<select  name=key  size=15>”)
    while (rs.next())
         menu.append(“<option>  ” + rs.getString(1)) ;
    menu.append(“</select>”) ;
    return menu.toString() ;
}
```

Remarks

- This works, but as observed there is a certain loss of modularity. It seems odd that, e.g., the name of a form parameter, **key**, is defined in this layer.
- The **getMenu()** method can throw an exception.
- This is allowed. There is a mechanism at the JSP level for handling such errors.

Exception Handling in JSP

- The JSP **page** directive has an optional attribute called **errorPage**. In our case we will include the directive:

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

- If this attribute is defined, its value must be the URL of another JSP page.
- If an uncaught exception occurs in the servlet, control will be forwarded to the specified page.
- That page should include the following directive:

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

  asserting that it is an error-handling page.
- Scriptlets in such pages have access to an extra predefined variable called **exception**.

Viewing a Student Entry

- The **action** attribute in the first form sends the browser to a JSP page called **view.jsp**, setting a parameter called **key**.
- The trick for persuading the bean to look up the corresponding entry in the database will be to set a write-only trigger property, also called **key**.
- Here we can use the feature of **jsp:setProperty** that if a value is not explicitly specified in the tag, the action looks for it in a request parameter of the same name.
- The element in the JSP page is thus:

```
<jsp:setProperty  name="dbs"  property="key" %>
```

The **DBSessionBean setKey()** Method

```
public void setKey(String key) throws SQLException {
    ResultSet rs = stat.executeQuery(
                        "SELECT * FROM " + table + "  " +
                        "WHERE login=' " + key + " ' ") ;
    if(rs.next()) {
        login       = rs.getString(1)
        lastname    = rs.getString(2) ;
        firstnames = rs.getString(3) ;
        email       = rs.getString(4) ;
        dept        = rs.getString(5) ;
    } else
        throw new SQLException("DBSessionBean: Record not
  found") ;
     this.key = key ;
}
```

Remarks

- This methods sets the 5 private **String** variables **login**, **lastname**, **firstnames**, **email**, **dept**.
- It also saves the value of **key**.

Displaying the Fields

- The bulk of the page **view.jsp** is now just an HTML form with 5 input elements. . .

```
<form method=post  action=update.jsp>
Login ID:
    <input  type=text  size=10  name=login
     value='<jsp:getproperty name="dbs"  property="login"/>'
   >
. . . 4 more input elements for the other fields . . .
</form>
```

- For this to work, the Bean class must have **get** methods for the 5 fields, **login**, **lastname**, **firstnames**, **email**, and **dept**.
- One slightly odd feature is the nesting of the XML element within an HTML tag.
  - One could put the initial text in the body of a small **textarea** element instead.

The Update Page

- The last page is the page **update.jsp**, which is responsible for updating a row in the table.
- Although it doesn't generate any output, it illustrates several features, so we reproduce it in full. . .

The Page **update.jsp**

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

<jsp:useBean id="dbs"  class="DBSessionBean”
    scope="session">
      <jsp:forward page="timeout.html” />
</jsp:useBean>

<jsp:setProperty  name="dbs"  property="*"/>

<jsp:setProperty name="dbs"  property="saved"  value="true"/ >

<jsp:forward page="select.jsp"/>
```

Remarks

- In this case, *if* the **jsp:useBean** action must create a new session bean, it presumably means the old session timed out.
  - The body of the action transfers control to an error page.
- If the property name in a **jsp:setProperty** tag is **"*"**, the action looks for all parameters of the request whose names that correspond to names of Bean properties, and copies parameter values to Bean properties.
  - In this case it will automatically set **login**, **lastname**, **firstnames**, **email**, and **dept**, which are assumed to have **set** methods in the Bean class.
- The property **saved** is a write-only trigger property, which causes the current field values to be written to the row with the currently established key.
- Finally the page transfers control back to **select.jsp**.

The DBSessionBean setSaved() Method

```
public void setSaved(boolean saved) throws SQLException {
    if(saved)
        stat.executeUpdate(
            "UPDATE " + table + " SET " +
            "login=’ "       + login       + “', " +
            "lastname=’ "    + lastname    + “', " +
            "firstnames=’ "  + firstnames  + “', " +
            "email=’ "       + email       + “', " +
            "dept=’ "        + dept        + “' " +
            "WHERE login=’ " + key + “'”
        ) ;
}
```

Lessons

- The set of standard actions provided by JSP is minimalist (6 tags), but the example suggests they are quite flexible, and can implement applications with non-trivial logic.
  - We needed very few Java scripting inserts.
- Session tracking and error handling are particularly elegant.
- Caveats:
  - The lack of iteration in the standard actions means that sometimes the Bean has to take over HTML generation, e.g. for tables and menu elements.
  - The trigger+linked properties approach works, but seems slightly contrived for complex operations?

A Quick Tour of Tag Libraries

Motivations

- In the last two lectures we discussed a particular approach to authoring dynamic Web content: JSP standard actions together with JavaBeans.
- We argued that this approach had important advantages over the scripting approach.
  - Free use of Java scripting elements in the body of the HTML document is confusing.
- At its best, the action-based approach cleanly separates the presentation aspects of the document from the computational logic.
- However we also saw that the standard actions are fairly limited.
  - They sometimes, for example, force one to generate fragments of dynamic HTML inside the Bean. This weakens the desired separation.

Customized Actions

- The solution in JSP 1.1 is allow the creation of customized actions, or tag libraries.
- A customized action can accept multiple arguments (as attributes) from the HTML document.
  - We no longer have to shoehorn all functionality into the get and set methods associated with JavaBean properties.
- Even more interesting, a customized action element can do non-trivial processing on the text in its body.
- In particular, a customized action element can yield multiple copies of the text in its body.
  - *Iterative* processing!
  - It can also choose to ignore the body text: *conditonal* processing.

Hello World

- Here is a JSP document that uses a tag library:

```
<%@  taglib uri="hellolib.tld"  prefix="test" %>
<html><head></head><body>
<test:hello  user="Bryan" />
</body></html>
```

- When I visit this page, the unexciting response is:

  – Of course the excitement is that this was done with a custom tag, **test:hello** .

The Tag Library Directive

- The **jsp:taglib** directive informs the JSP container that this page will be using a library of customized tags.
- The most important attribute of the directive is **uri**.
- The value of this directive is a URL referencing the Tag Library Descriptor file (TLD).
- This is an XML document describing the syntax of the custom elements, and specifying where to find the associated tag handler classes.
- The **prefix** attribute just specifies a prefix that will be appended to the tag names within this document.
  - The prefix is not part of the library definition—it is simply a convenience to avoid name classes between tags used in a page.

The Tag Library Descriptor File

- The TLD file is in XML format.
- It has some preamble describing the XML version and document type.
- The main body is a **taglib** element.
- This element in turn contains some global information about the library, then (most importantly) a series of **tag** elements.
- Each **tag** element defines the syntax and (indirectly) the semantics of a custom tag:
  - Tag name.
  - Handler class.
  - Attribute names and properties.
  - How to deal with the element body, if there is one.

TLD File for the “Hello” Example

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib
    PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library
  1.1//EN"
    "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">
<taglib>
   <tlibversion>1.0</tlibversion>
   <jspversion>1.1</jspversion>
   <shortname>mytags</shortname>
   <uri></uri>
   <info> My first tag library  </info>
   . . . tag element(s) . . .
</taglib>
```

The TLD **tag** Element

```
<tag>
    <name>hello</name>
    <tagclass>mytags.HelloTag</tagclass>
    <info>  Says "Hello"  to a user.  </info>
    <attribute>
        <name>user</name>
        <required>true</required>
        <rtexprvalue>true</rtexprvalue>
    </attribute>
    <bodycontent>EMPTY</bodycontent>
</tag>
```

Remarks

- The preamble defines the XML version and an XML Document Type Definition file for the document.
- *Required* elements inside a **taglib** element are
  - the **tlibversion** element which give a version number for the tag library, and
  - the **shortname** element used to identify the library.
- A **tag** element specifies:
  - Java classes that define the behavior of the action.
  - **attribute** elements that define the tag's attributes.
  - An attribute may be "**required**" or optional.
  - Its value may be required to be a constant string, or allowed to be a dynamic, request-time expression ("**rtexprval**" element).
  - The *body content* of the the defined element may be specified to be **EMPTY**.
  - It may also be specified to be **JSP**, or **tagdependent**.

Tag-Handling Classes

- In the example, the tag is handled by a class called **HelloTag** which belongs to a package called **mytags**
  - Some JSP containers require that tag-handling classes be defined in a package.
- A tag handling class must implement the interface **javax.servlet.jsp.tagext.Tag**.
- In practice the easiest thing is to extend the class **javax.servlet.jsp.tagext.TagSupport**, which provides default implementation of various methods.

Handling Class for the **hello** Tag

```
package mytags ;
import javax.servlet.jsp.* ;
import javax.servlet.jsp.tagext.* ;
import java.io.* ;
public class HelloTag extends TagSupport {
   public  int  doStartTag()  throws JspException {
      try {
         JspWriter out = pageContext.getOut() ;
         out.print("<h1>Hello " + user + "!</h1>") ;
      } catch (IOException e) {
         throw new JspTagException(e.getMessage()) ;
      }
      return SKIP_BODY ;
   }
   public void setUser(String user) {
      this.user = user ;
   }
   private String user ;
}
```

Remarks

- When a custom tag is encountered in a JSP page, **set** methods are invoked on the handler for each of the attributes.
  - In this case there is one attribute: **user**.
- The **doStartTag()** method is then invoked.
- In general this can process the attributes, generate some output, and decide what should be done about the body of the element, if any.
- In this case there is no body. The method simply returns the value **Tag.SKIP_BODY**.
  - The other possibility for a simple elements is **Tag.EVAL_BODY_INCLUDE**, which tells the JSP container to process the body in the usual way, outputting the results as it goes.

Actions that Process their Bodies

- More sophisticated custom actions may need to do some non-trivial, tag-dependent processing on the text in their bodies.
  - Rather than leaving the body to default JSP processing.
- Tag handling classes for such actions should implement the more complex **javax.servlet.jsp.tagext.BodyTag** interface.
  - Or extend the associated **javax.servlet.jsp.tagext.BodyTagSupport** class.
- Actions handled by these classes can also *process their bodies repeatedly*, thus generating *iterative output*.

An Iterative Element

```
<%@ taglib uri="selectlib.tld" prefix="test" %>
<html><head></head><body>
<table border cellspacing=0 cellpadding=5>
<tr bgcolor=lightblue>
    <td>Login id</td> <td>Last name</td> . . . <td>Dept</td>
</tr>
<test:select session="dbs" columns="*" table="it1fall00">
   <tr>
      <td> <test:field column="1"/> </td>
      <td> <test:field column="2"/> </td>
      <td> <test:field column="3"/> </td>
      <td> <test:field column="4"/> </td>
      <td> <test:field column="5"/> </td>
   </tr>
</test:select>
</table>
</body></html>
```

The Displayed Page

Login id	Last name	First names	email	Dept
wcao	Cao	Wenzhong	wcao@cs.fsu.edu	CSIT
flora	Flora	Albertine Mary	flora@eng.fsu.edu	EE
Fulay	Fulay	Amit	fulay@cs.fsu.edu	CS
srikanth	Garimella	Srikanth	garimell@eng.fsu.edu	EE
gao	Gao	Dongmei	gao@cs.fsu.edu	CS
goregaok	Goregaoker	Sachin	goregaok@cs.fsu.edu	CS
gou	Gou	Jihua	gou@eng.fsu.edu	IE
jiejiang	Jiang	Jie	jiejiang@cs.fsu.edu	CS
tiejiang	Jiang	Tiehu	hu@eng.fsu.edu	CS
Kiran	Kaja	Kiran P	kiran@cs.fsu.edu	CS
mkotturu	Kotturu	Madhavi	kotturu@cs.fsu.edu	CS
Wei2	Li	Wei	wei2@cs.fsu.edu	CS
pant	Pant	Saurabh	pant@cs.fsu.edu	CS
patel	Patel	Nikhil M	patel@cs.fsu.edu	CS
srr9235	Ramaswamy	Shanmuga Raja	ramaswam@cs.fsu.edu	CS
windsnow	Rao	Xi	xi@cs.fsu.edu	CS
jshang	Shang	Jin	shang@cs.fsu.edu	CS
anjan	Srinivas	Anjan	anjan@eng.fsu.edu	EE
valsalak	Valsalakumari	Lakshmi	valsalak@cs.fsu.edu	CS
yilewang	Wang	Yilei	yilewang@cs.fsu.edu	CS
gu	Yan	Wenchang	wenchyan@cs.fsu.edu	CS
zyr982	Zhang	Liangying	liazhang@cs.fsu.edu	CS
jin0328	Zhang	Wenjin	wenzhang@cs.fsu.edu	CS
zhao6930	Zhao	Dangzhi	dzz5355@garnet.acns.fsu.edu	CS
zheng	Zheng	Wei	zheng@cs.fsu.edu	CS

Remarks

- The **select** action from **selectlib.tld** executes an SQL **SELECT** query on some database.
- The body of the element is processed once for every row returned by the query.
- In our page, each evaluation of the body generates one row of an HTML table (this is not part of the library.)
- The **field** action from the **selectlib** library returns the column value with index specified by the **column** attribute.
- Now all HTML generation is handled in the JSP page—the Java code is only responsible for accessing the data base and managing iteration over the result set.
- A similar library could be used to generate the HTML menu in the example from the previous lecture.

Handling Class for the **select** Tag

- Apart from attribute setting method, the handling class, **SelectTag**, now defines two methods: **doStartTag()** and **doAfterBody()**.
- *Both* of these return an **int** value:
  - for a class implementing **BodyTag**, possible return values are **SKIP_BODY** or **EVAL_BODY_TAG**.
- As before, if **doStartTag()** returns **SKIP_BODY**, the body is skipped; if it returns **EVAL_BODY_TAG**, the body is processed.
- But now a **doAfterBody()** method, called after the body is processed, may *also* return **EVAL_BODY_TAG**.
  - If it does so, the body is processed *again*. This continues until **doAfterBody()** finally returns **SKIP_BODY**.

The **SelectTag doStartTag()** Method

```
public  int  doStartTag()  throws JspException {
    try {
        HttpSession session = pageContext.getSession() ;
        DBSession dbs = (DBSession)
    session.getAttribute(sessionID) ;
        if(dbs == null) {
            dbs = new DBSession() ;
            session.setAttribute(sessionID, dbs) ;
            . . .
        }
        rs = dbs.stat.executeQuery("SELECT " + columns +
                                   "  FROM "  + table) ;
        if (rs.next())
            return EVAL_BODY_TAG ;
        else
            return SKIP_BODY ;
    } catch ( . . . ) { . . . }
}
```

Remarks

- The method starts with some boilerplate, session-tracking code.
  - We recycle the **DBSession** from the servlet version of the student database example.
  - The **pageContext** instance variable allows us to retrieve JSP predefined variables like **session**.
- The instance variables **sessionID**, **columns** and **table** correspond to the tag attributes.
  - They are initialized in **setSession()**, **setColumns()** and **setTable()** methods, not reproduced here.
- The main business is in the **executeQuery()** method. Its result is placed in another instance variable, **rs**.
- The return value of **doStartTag()** is determined by **rs.next()**.
  - On its first call, this returns **true** iff the result set is not empty.

The **SelectTag doAfterBody()** Method

```
public  int  doAfterBody()  throws  JspException {
   try {
      if(rs.next())
         return EVAL_BODY_TAG ;
      else {
         BodyContent body = getBodyContent() ;
         body.writeOut(getPreviousOut()) ;

         return SKIP_BODY ;
      }
   }  catch ( . . . ) { . . . }
}
```

Remarks

- If **rs.next()** is **true**, this does nothing but return **EVAL_BODY_TAG** again—the iteration continues.
- If **rs.next()** is **false**, the result set has been exhausted.
- Before returning the value **SKIP_BODY**, which tells the JSP container *not* to process the body again, the method outputs the accumulated result of (repeatedly) processing its body.
  - Any tag handler that implements **BodyTag** has accepted this responsibility.
  - Details are beyond the scope of this lecture, but the code given here should work when the JSP default processing mode is adequate for the body content.

Handling Class for the **field** Tag

- The handling class, **FieldTag**, does not need to process a body.
- It extends **TagSupport**, and defines the method **doStartTag()** (plus a **set** method for the **column** attribute).
- The crucial new feature here is the call to **findAncestorWithClass()**.
- It returns the tag-handling object for the most closely enclosing JSP element with specified tag-handling class.
- In our case this will be the tag-handling object for the surrounding **select** element.
  - The **rs** field of that object is the **ResultSet** from which the column value should be extracted.

The **FieldTag doStartTag()** Method

```
public int doStartTag() throws JspException {
    try {
        JspWriter out = pageContext.getOut() ;
        SelectTag parent =
          (SelectTag) findAncestorWithClass(this, SelectTag.class)
   ;
        out.print(parent.rs.getString(Integer.parseInt(column))) ;
    } catch (. . .) { . . . }
    return SKIP_BODY ;
}
```

Remarks

- We can reuse exactly the same tag library—consisting of **select** and **field** elements—to build an HTML menu of keys from the table:

```
<%@  taglib  uri="selectlib.tld"  prefix="test"  %>
. . .
<select name=key size=15>
   <test:select session="dbs"  columns="login"
  table="it1fall00">
      <option> <test:field  column="1" />
   </test:select>
</select>
```

- This degree of reusability reflects an excellent level of separation between presentation logic and application logic.
  - The price is we have to implement our own library—not in general an easy thing!

Menu Generated Using **selectlib.tld**