Pure server-side Web Applications with Java, JSP

- Discussion of network-level http requests and responses
- Using the Java programming language (Java servlets and JSPs)
- The role of application servers
  - The "OS" of web apps
  - Data contained in app server’s request, session, application scopes

Application Servers: the Essential Tool of Server-Side Programming

- Java servlet containers, responsible for
  - facilitating the http communications
  - Providing web app context
  - ...
- May also (but not necessarily) operate as web servers, i.e., serve static pages
- Tomcat is an app server and the reference implementation of the Java servlet and JSP specifications
  - Also serves static pages

Install and Check Tomcat

Discussion session
Installing Tomcat

- Install stable production release
  - Do not install alpha, beta, “milestone”, or “nightly” builds
- You need a compatible J2SE or J2SDK
- If installed in directory X, set environment variable JAVA_HOME to X
- Use self-extracting .exe and follow directions
- Set CATALINA_HOME to directory where Tomcat is installed

Starting and Testing Tomcat

- Start Tomcat using bin/startup.bat or “Start Tomcat” icon in program group
  - Preferably do not set up Tomcat as an “automatic start” service
- Browse to http://localhost:8080/
  - You should see Jakarta project home page
  - If failure, come to discussion
- Run http://localhost:8080/examples/jsp/dates/date.jsp

HTTP Requests and Responses
HTTP Basics

- TCP/IP protocol used by Web servers, clients
- Synchronous
  - i.e., client sends request waits for response
- Stateless
  - i.e., all info needed by server-side must be contained in http request
  - Using appropriate session management techniques app servers go around restrictions of statelessness
- We show next the request and response message strings that go back and forth in interactions
  - Only for educational purposes.
  - You will never code such strings directly. App server will do it for you.

http too slow?

- Yes
- We will later discuss websockets, in the context of content-rich and live visualizations

Syntax of an HTTP Request

- `<method> <request URI> <HTTP-version>`
  - Important ones: GET & POST
  - Other methods: HEAD, PUT, DELETE, CONNECT, OPTIONS, TRACE
- Header fields
  - `Accept: text/html, text/xml, ...`
  (acceptable response types)
- Message body (optional) (after blank line)
Example HTTP request

GET / HTTP/1.1
Host: www.db.ucsd.edu
User-Agent: IE/9.0
Accept: text/html, text/xml
...

Syntax of an HTTP response

- Reminds email syntax
- `<HTTP-version> <status-code> <reason>`
  - E.g., status codes from 500-599 indicate server-side errors
- Header fields
  - `Content-Type: text/html (or other type)`
- Message body (optional) (after blank line)

Communicating Parameters from Browser to Server

- Overview of the "multiplier" application
  - First assuming servlet at server
    - The shown servlet use is not directly supported but is instructive
    - We will replace the servlet with a jsp later
  - The browser-to-server communication aspects remain the same regardless of direct servlet or jsp use

Entry

multiplier.html

Submission of form

servlet/MyMultiplier

Entering "2" and submitting caused http request ".../servlet/MyMultiplier?num=2"
We refer to num=2 as request parameter
Issuing a /servlet call directly is not supported because it is a security liability. For instruction purposes, let us assume it is supported.

Communicating Data Provided in Forms: GET, POST and parameters

- The HTML of multiplier.html

```html
<html>
<head>
<title>Multiplier Form</title>
</head>
<body>
Welcome to the page that helps you multiply times 3
<p>
<form method="GET" action="servlet/MyMultiplier">
Provide the number to be multiplied:
<input type="text" name="num"/>
<p>
<input type="submit" value="Click Here to Submit"/>
</form>
</p>
</body>
</html>
```

If you are not fluent HTML try to write your resume in HTML using just a text editor.
POST vs GET (mechanics)

- Upon submitting "2" the browser emits URL
  - GET /multiplier/servlet/MyMultiplier?num=2 HTTP/1.1
    Host: localhost:8080

- If HTML form may create more than 255 characters use <FORM METHOD=POST>
  - Form data will be in body of http request
    - POST /multiplier/servlet/MyMultiplier HTTP/1.1
      Host: localhost:8080
      num=3

GET vs POST; when to use

- GET
  - Supposed to retrieve a resource from the server
  - Bookmarking friendly
  - Easy fiddling of parameters (security hole)
  - Caching bug friendly (more later)

- POST
  - Supposed to change server state
  - Can transmit far larger arguments
  - Arguments not displayed on the URL
  - On refresh, browser says “data will be POSTed again”

More Input Forms: Dropdown menus

```html
<html>
  <head><title>Multiplier Form</title></head>
  <body>
    Welcome to the page that helps you multiply times 3 using a dropdown menu<p>
    <form method="GET" action="servlet/MyMultiplier">
      Provide the number to be multiplied:
      <select name="num">
        <option value="1">One</option>
        <option value="2">Two</option>
      </select><p>
      <input type="submit" value="Click Here to Submit"/>
    </form>
  </body>
</html>
```
Encoding URIs

- HTTP only permits letters, digits, underscores and a few more
- Browsers take care of "special" symbols, using the RFC2277 encoding

Example of Encoding Characters in a URI Using the RFC2277

- Consider a page asking for emails

  <HTML> <TITLE>Email Submit Page</TITLE> <BODY>
  <FORM METHOD=GET
     ACTION=http://gyro.ucsd.edu:8080/subemail.jsp>
    Type your e-mail here:
    <INPUT TYPE="text" NAME="eml"/> <P>
    <INPUT TYPE="SUBMIT" VALUE="Click Here"/>
  </FORM> </BODY> </HTML>

- User types yannis@cs.ucsd.edu
  - GET http://subemail.jsp?eml=yannis@cs.ucsd.edu HTTP/1.1
  - Host: gyro.ucsd.edu:8080
Servlets: The Assembly Language of Java-based Web Server-Side Programming

Java-Based Server-Side Programming 101: Servlets

- **Servlet**: Java program run inside the app server (Tomcat in 135)
- **Inputs** http requests
  - App server provides request data to servlet in appropriate object format
- **Typically** (but not necessarily) return http responses of html content type

Multiplication example revisited: Browser -> App Server -> Servlet

- Create Web app (directory) `multiplier` under `webapps`
- Place `multiplier.html` in it
- Browse to `http://localhost:8080/multiplier/multiplier.html`
- When form is submitted browser issues http GET request
  - ACTION specifies URL to be invoked
  - URL of servlet may be relative (as in example)
    - "servlet" is not directory; simply indicates it is servlet
  - Or absolute (would be `http://localhost:8080/multiplier/servlet/MyMultiplier`)
  - recall, for security reasons the "/servlet" request will not go through – theoretically, assume it does
Making your app available to the world via standard http port 80

- Default http port is 80 (not 8080)
- Set via tomcat
- If your machine has domain name my135machine.org and the appropriate access rights then any browser from the internet may issue:
  http://my135machine.org:8080/multiplier/multiplier.html
- any prior mention of localhost:8080 should be replaced with my135machine.org

Multiplication example revisited:

Browser -> App Server -> Servlet

- Application server knows where servlet code MyMultiplier.class resides
- Activates MyMultiplier.class, passing the request parameters in object format
- MyMultiplier.class prints html in the http response
  - Details coming up
- Next: The Java code of MyMultiplier.java

```java
import java.io.*;
import java.text.*;
/* following packages encapsulate Servlet API */
import javax.servlet.*;
import javax.servlet.http.*;

public class MyMultiplier extends HttpServlet {
    /* Overrides doGet coming with HttpServlet */
    public void doGet(HttpServletRequest req, HttpServletResponse res)
            throws ServletException, IOException {
```
res.setContentType("text/html");

PrintWriter out = res.getWriter();

out.println("<HTML><HEAD><TITLE>
Multiply times " + 3 + "</TITLE></HEAD>"
);
out.println("<BODY>");
String parameter = req.getParameter("num");
/* Ignoring the possibility that parameter is not integer */
out.println(parameter + " * " + 3 + " = " +
3 * (Integer.parseInt(parameter)));
out.println("</BODY>");
out.println("</HTML>");
}

Compiling & Deploying the Servlet

• Compile MyMultiplier.java
  – Include in CLASSPATH environment variable
    <CATALINA_HOME>\common\lib\servlet.jar
• Place MyMultiplier.class in an appropriate location where Tomcat expects to find it
  – We skip this detail since we won’t actually write and deploy servlets
• Restart Tomcat

Servlet Life Cycle

• First time a servlet is called:
  – init() method is called
    • Normally provided by HttpServlet
    • Unless you want to set up resources that exist for the whole lifetime of the servlet (rare)
    • Object (extending HttpServlet) is instantiated and becomes memory resident from now on
    • Class variables exist for entire life of object
• Series of GET, POST, … HTTP calls lead to doGet(), doPost(), etc calls to the object
• Servlet removed with destroy()
  – Tomcat may call destroy() any time
  – you may write your own destroy() to save state upon receiving destroy()
Handling POST Method Calls

• Whether parameters are communicated by GET or POST is normally irrelevant to your code
• However you have to provide (override) doPost() of HttpServlet

```java
public void doPost(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException {
    doGet(req, res);
}
```

Handling the Other Method Calls

• DELETE, HEAD, OPTIONS, PUT, TRACE
• Corresponding doDelete(), doHead(), etc
• Normally developer does nothing
• HttpServlet provides defaults

What is Wrong with Servlets

• The "look" of the resulting HTML is buried in println() statements
• Web designers cannot work this way
• Business logic and presentation horribly mixed
• other issues...
Java Server Pages: Embedding Java Code in Static Content

Why JSPs?

• Need to separate
  – the business logic implementation
    • done by web developer
  – from implementing the look-and-feel
    • done by web designer

The Key Idea Behind JSPs

• HTML page with embedded Java code (in the form of JSP elements)

  <HTML>
  <HEAD>
    <TITLE>Date JSP (Textbook Listing 5.1)</TITLE>
  </HEAD>
  <BODY>
    <BIG>
      Today's date is <%= new java.util.Date() %>
    </BIG>
  </BODY>
  </HTML>
Deploying JSPs

- JSP file has .jsp suffix
- Store JSP file (in text) in app directory
  - Recall, under webapps
- Invoke as
  \texttt{http://<host>/<web-app>/<file>.jsp}

Compilation

- At first access of JSP
  - Jasper translator generates Java servlet code
    - Loads in \texttt{<CATALINA_HOME>/work/Standalone/<host>}
    - Jasper compiler generates Java Servlet class file
      - Loads in same directory

```java
package org.apache.jsp;
/* Automatic Imports */
import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;
import org.apache.jasper.runtime.*;

public class date_jsp extends HttpServlet {
    private static java.util.Vector _jspx_includes;

    public java.util.List getIncludes() {
        return _jspx_includes;
    }

    /* Similar to doGet() */
    public void _jspService(HttpServletRequest request,
                            HttpServletResponse response)
        throws java.io.IOException, ServletException {
```

Implicitly Declared Objects

- You may use the following objects in the Java code of your JSP
  - **request**: well-known HttpServletRequest object
    - transfers parameters
  - **response**: still important for writing non-body fields of HTTP response
  - **session**: maintain parameters accessed by all steps of a session
    - Very important, we’ll come back to it
  - **application**: maintain parameters accessed by all jsp’s of a web application

```java
/* Implicit objects defined next */
JspFactory _jspxFactory = null;
javax.servlet.jsp.PageContext pageContext = null;
HttpSession session = null;
ServletContext application = null;
ServletConfig config = null;
JspWriter out = null;
Object page = this;
JspWriter _jspx_out = null;

try {
  /* Initialization of implicit objects */
  _jspxFactory = JspFactory.getDefaultFactory();
  response.setContentType("text/html;charset=ISO-8859-1");
  pageContext = _jspxFactory.getPageContext(this, request, response,
                             null, true, 8192, true);
  application = pageContext.getServletContext();
  config = pageContext.getServletConfig();
  session = pageContext.getSession();
  out = pageContext.getOut();
  _jspx_out = out;
  
  /* Output of HTML code of jsp */
  out.write("<HTML>
  
  <HEAD>
  <TITLE>Date JSP (Textbook Listing 5.1)
  </TITLE>
  
  <BODY>
  <BIG>
   Today's date is 
   
  </BIG>
  
  </BODY>
  
  </HTML>
  
  
  try {
    /* Initialization of implicit objects */
    _jspxFactory = JspFactory.getDefaultFactory();
    response.setContentType("text/html;charset=ISO-8859-1");
    pageContext = _jspxFactory.getPageContext(this, request, response,
                   null, true, 8192, true);
    application = pageContext.getServletContext();
    config = pageContext.getServletConfig();
    session = pageContext.getSession();
    out = pageContext.getOut();
    _jspx_out = out;
    
    if (ref = null) out.write("Ref is null: ");
    out.write("Ref is not null: ");
    if (this == null) out.write("This is null: ");
    out.write("This is not null: ");
    
    catch (Throwable t) {
      out.write("Exception handle ");
      out.write(Date.class.getName());
      out.write(" handle ");
      out.write(Exception.class.getName());
      
      finally {
        if (this == null) _jspxFactory.releasePageContext(pageContext);
      }
    }
  }
```
JSP Elements

- **JSP Directives**
  - Includes, imports, etc
- **JSP Scripting Elements**
  - Java code, expressions, variable declarations
- **JSP Action Elements**
  - Beans, tag libraries, etc
  - We’ll discuss later

JSP Directives

- `<%@ directive { attr="value" }%>`
- `<%@ include file="file.html" %>`
- `<%@ page import="package name" %>`

```html
<HTML>
<HEAD>
<TITLE>dateWithImport.jsp</TITLE>
</HEAD>
<BODY> <BIG>
<%@ page import="java.util.*" %>
Today's date is <%= new Date() %> 
</BIG> </BODY>
</HTML>
```

- Recall: some packages automatically imported

JSP Scripting Elements

- **Expressions**
  - `<%= Java_expression %>`
  - Example: `<%= i+1 %>`
  - Evaluates expression, casts into String, places in output
- **Scriptlets**
  - `<% Java_code %>`
  - Example:
    ```java
    <% int times ;
    times = 3 ; %>
    ```
  - Code inlined in `_jspService()`
- **Scriptlets have semicolons, expressions don’t**
Multiplier example, revisited in jsp

multiplier.html

<html>
<head>
<title>Multiplier Form</title>
</head>
<body>
Welcome to the page that multiplies by 3
<p>
<form method="GET" action="multiplyJSP.jsp">
Provide the number to be multiplied:
<input type="text" name="num"/>
<p>
<input type="submit" value="Click to Submit"/>
</form>
</body>
</html>

multiplyJSP.jsp

<html>
<head>
<title>Multiply by 3 (JSP)</title>
</head>
<body>
<%
    int times = 3;
    String param = request.getParameter("num");
%>
<%= param + " * " + times + " = " + times *
(Integer.parseInt(param)) %>
</body>
</html>
Two kinds of declarations in JSP Scripting Elements

- **Local variables simply part of scriptlets**
  - See code of
    `<CATALINA_HOME>/work/Standalone/localhost/jmultiplier/jmultiply_jsp.java`

- **Class variables (not in \_jspService())**
  - `<! int times ; %>`
  - See `jMultiplyWithClassVariable.jsp`
  - If we have in JSP scriptlet
    `<% times = times + 1; %>`
  - It will be incremented every time JSP is called
    - from same or different sessions

Deployment Descriptor and URL Mapping

- **Provide configuration/deployment information in WEB-INF/web.xml**

- **Uses URL mapping**
  - Say, you do not want users to know that you use jsps and the names of your code files
  - maps the jsp's actual name to a URL pattern

- **Can access jsp by**

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app>
  <!-- General Web Application Description -->
  <display-name>Multiplier built as JSP</display-name>
  <description>Multiplies times 3, using JSP</description>
  <servlet>
    <servlet-name>multiplier</servlet-name>
    <jsp-file>multiplyJSP.jsp</jsp-file>
  </servlet>
  <servlet-mapping>
    <servlet-name>multiplier</servlet-name>
    <url-pattern>/multiply</url-pattern>
  </servlet-mapping>
</web-app>
```
Servlet Context Path

- Default context name of Web application is the name of the webapps subdirectory
  - in running example, multiplier
- Create alias context name if you want to hide the subdirectory name
- Add Context element in conf/server.xml, inside
  `<Host name="localhost" ...>
  <Context path="/mult" docbase="multiplier"/>
- Path is matched against URLs’ beginning
  - must be unique
  - Try http://localhost:8080/mult/multiply?num=10

Automatic Reload

- Default configuration does not check whether files are replaced
  - Appropriate setting in production mode
- We can avoid stopping and restarting Tomcat during development/compilation
- by enabling automatic reloading
  - to effect for an individual web app edit
    - server.xml and add reloadable attribute
  - `<Context ...this web app"... reloadable="true"/>
- To effect automatic reload for all applications add
  - `<DefaultContext reloadable="true"/>

Deployment Revisited

- All uses of servlet names also apply to JSP’s
  - Eg, you may not want someone to know that you have used (a particular) .jsp to implement your page and you want to use URL mapping to hide name
- Declaration of name almost same with servlets
  `<servlet-name>Multiplier</servlet-name>
  `<jsp-file>jmultiplier.jsp</jsp-file>
Scope Issues in JSPs

How to store data in the app server and control which http requests get access to them

- **Application**
  - Information accessible/shared by all requests of same application (same app context)
- **Session (most important)**
  - Session: Set of requests from same browser process
    - Browser windows may be in same process
  - Share information within session
  - Non-obvious how given HTTP statelessness
- **Request**
  - Share information within http request
- **Page (almost useless)**

Application Level Attributes

- `application` implicit variable of JSP
- In servlet obtained by `application=getServletContext()`
- Exchange attribute info across all calls
  - `application.getAttribute(name)`
  - `application.setAttribute(name, object)`
  - Can do the same with class variables
  - Or with a database
    - At higher cost but with persistence
    - No synchronization and ACID properties
Counter Example

```html
<% Integer i =
(Integer)(application.getAttribute("counter"));
    if (i == null) { i = new Integer(0) ; }
    else { i = new Integer(i.intValue() + 1) ; }
application.setAttribute("counter", i) ;
%>
Your application has visited <%= i %> times this page.
</HTML>
```

Getting Web Application Initialization Parameters

- Define application initialization parameters in the deployment descriptor

```xml
<web-app>
  <!-- other stuff we've seen..>
  <context-param>
    <param-name>developer</param-name>
    <param-value>yannis@cs.ucsd.edu</param-value>
  </context-param>
  <!-- other stuff we've seen..>
</web-app>
```

- application.getInitParameter(name)

Session Level Attributes

- HTTP is stateless
- But your applications most often involve stateful sessions
- Session-level attributes pass data across the requests of a session
- App server provides implicit session object
- In servlets: req.getSession(), where req is the HttpServletRequest parameter
- Behind the scenes Tomcat employs cookies and/or URL rewriting to implement the session object
Maintaining Session Information with the Implicit `session` Object

```html
<HTML>
<HEAD>
<TITLE>Counter Web Application</TITLE>
</HEAD>
<BODY>
<% Integer i=(Integer)(session.getAttribute("counter"));
    if (i == null) { i = new Integer(0); }
    else { i = new Integer(i.intValue() + 1); }
    session.setAttribute("counter", i);
%>
Your session has visited <%= i %> times this page.
</BODY>
</HTML>
```

Session Duration

- Session data are automatically deleted after
  - client is inactive for a period
  - Tomcat default is 30 minutes
  - call of `HttpSession.invalidate()`
- Dynamic reset of session duration with `HttpSession.setMaxInactiveInterval()`
  - In seconds
- Set the default for all web applications following path
  `web-app/session-config/session-timeout` in
  `<CATALINA_HOME>/conf/web.xml`

Other Methods of passing Information
Direct Use of the `response` Object

- Set values for various headers
  - `response.setContentType(String <MIME type>)`
- Add extra HTTP headers
  - `addHeader(java.lang.String name, java.lang.String value)`
  - Other "versions" for int, Date, etc types
- Add cookies (discussed next)
- Send error responses

Cookies

- Way to store information on the client side
- Server includes `Set-Cookie` header
  - Eg, `Set-Cookie: multiply$Id=+$7BE2; path=/`
  - Implicitly associated with URL of server that provided
  - Explicitly associated with provided path
- Web client stores on cookie repository
  - if cookies from this site are enabled
  - Until expiration
    - Default is the browser session

Cookies (cont'd)

- When web client makes subsequent http request to domain/path all matching cookies are attached
  - Eg, `Cookie: multiply$Id =+$7BE2`
- Constructor
  `javax.servlet.http.Cookie(String name, String value)`
  `response.addCookie(Cookie value)`
  `request.getCookies()` returns `Cookie[]`
- Bunch of setter methods for changing default path, id, lifetime properties of cookie
When Should One Use Cookies?

- Use cookies if
  - No confidential info is released
  - You have to utilize their longevity
    - Cookies that live across browser startup/shutdown
  - Web app does not fall apart if cookies are disabled by client
- Example: preset some forms
- Do not use for standard session management aspects

Hidden Fields

- Passing (non-user input) information across requests
- You need an HTML form to be present
  - Not applicable with HTML links
- `<INPUT TYPE="HIDDEN" NAME="<parameter>" VALUE="<value>"`>
- Prefer POST forms if you need to hide the hidden field from the URL
- Database keys are typical hidden fields
  - Example in databases section.

Putting it all together with the students example

- A "database" of students maintained in an application-scoped attribute "database"
  - "database" is a Java Map structure

Highlight points:
- "Model 1" programming paradigm
- **Extensive use of hidden id’s to capture**
  - which one of the many forms of the page has been submitted
  - which one of the many links has been clicked
“Model 1” programming

A quick way to code web applications with simple control flow

Model 1

1. List the pages and how requests lead from calling pages to called pages
2. List request parameters
3. Take note of which requests may transfer to a called page (called.jsp)
4. Start called.jsp with scriptlet code that collects its request parameters, figures out what request was made and makes the appropriate side effects on session and the database

calling.jsp (or calling.html)

```
<form action="called.jsp">
  <input type="hidden" name="n1" value="p1">
  <input type="hidden" name="n2" value="p2">
  <input type="submit" value="Submit">
</form>
```

Simplest case:
The called.jsp is accessible via a single calling.jsp
AND (simplifying assumption) we do not care about the possibility that the user may issue directly a call http://.../called.jsp

called.jsp

jsp scriptlet picks parameters p1, p2, ..., updating session & database

JSP expressions & scriptlets producing the html page
Think of enter.html as being the calling.html

Think of choose.jsp as being the called.jsp

Session

---

**enter.html**

```html
<html>
<head><title>Shopping app entry</title></head>
<body>
Welcome to the tiniest shopping app of the world:<p>
<form method="GET" action="choose.jsp">
Your name: <input type="text" size="20" name="user"/><p/>
<input type="submit" value="Proceed to Choose products"/>
</form>
</body>
</html>
```
choose.jsp

Choose.jsp: Scriptlet book - keeps data received from “calling” page

```html
<html>
<head><title>Order Page</title></head>
<body>
<% String user = request.getParameter("user");
session.setAttribute("user", user);
%>
Welcome <%= user %> <p>
<form method="GET" action="order.jsp">
Product:
<select name="product">
<option value="Notebook">Notebook</option>
<option value="Pen">Pen</option>
</select><p>
Amount: <input type="text" size="4" name="amount"/><p />
<input type="submit" value="Click to Order"/>
</form>
</body>
</html>
```

Choose.jsp: ... and then visualization part produces html

```html
<html>
<head><title>Order Page</title></head>
<body>
<% String user = request.getParameter("user");
session.setAttribute("user", user);
%>
Welcome <%= user %> <p>
<form method="GET" action="order.jsp">
Product:
<select name="product">
<option value="Notebook">Notebook</option>
<option value="Pen">Pen</option>
</select><p>
Amount: <input type="text" size="4" name="amount"/><p />
<input type="submit" value="Click to Order"/>
</form>
</body>
</html>
```
Now think of choose.jsp as being the calling.jsp

...and think of order.jsp as being the called.jsp

**enter.html**

**choose.jsp**

**order.jsp**

Session

———

`<html>
<head> <title>Provide credit card</title> </head> <body>

<% String product = request.getParameter("product");
    String amountAsString = request.getParameter("amount");
    Integer amount = Integer.parseInt(amountAsString);
    session.setAttribute("product", product);
    session.setAttribute("amount", amount);
%>

Dear <%= session.getAttribute("user") %>,
you have ordered <%= session.getAttribute("amount") %>
<%= session.getAttribute("product") %>
<p>
Please provide your credit card:<p>
<form method="POST" action="dummyCreditCharge.jsp">
<input type="text" size="16" name="credit" />
</form>
</body>
</html>`

**order.jsp: Again, first the scriptlet book keeps then the rest visualizes**
Next case:
Two (or more) pages (or generally forms) transferring to the same called.jsp
AND for some reason you need to know which one called

Distinguish which of the two (or more) requests invoked the called.jsp
JSP expressions & scriptlets producing the html page

One of the calling pages may be the called.jsp (the original use of Model 1)

Example: Student Table (download)
“Model 1” approach to Student Table

Model 1 pattern

```html
<html>
  <Initialization of "database" (students and nextPID)>
  <Retrieval Code (after initialization)>
    <% String action = request.getParameter("action"); %>
    <Insert Code>
    <Update Code>
    <Delete Code>
  </body>
  <table>
    <tr>
      <td><jsp:include page="menu.html"/></td>
      <td><Table Presentation Code></td>
    </tr>
  </table>
</html>
```

Retrieval code (after initialization)

```jsp
<-- Retrieval code (already initialized students and nextPID) -->
<% 
  // retrieves student data from application scope
  LinkedHashMap<Integer, Student> students =
  (LinkedHashMap<Integer, Student>)application.getAttribute("students");

  // retrieves the latest pid
  Integer nextPID = (Integer)application.getAttribute("nextPID");
%>
```
Insertion Code:

```java
// Check if an insertion is requested
if (action != null && action.equals("insert")) {
    // make new student to add to students map
    Student newStudent = new Student();

    // add the attributes from the request object to new student
    newStudent.setPID(Integer.parseInt(request.getParameter("pid")));
    newStudent.setFirstName(request.getParameter("first"));
    newStudent.setMiddleName(request.getParameter("middle"));
    newStudent.setLastName(request.getParameter("last"));

    // add new student to the map
    students.put(nextPID, newStudent);
    nextPID++;
    application.setAttribute("nextPID", nextPID);
}
```

Update Code

```java
// Check if an update is requested
if (action != null && action.equals("update")) {
    // get student that needs to be updated
    Student updateStudent = students.get(Integer.parseInt(request.getParameter("id")));

    // add the attributes from the request object to new student
    updateStudent.setPID(Integer.parseInt(request.getParameter("pid")));
    updateStudent.setFirstName(request.getParameter("first"));
    updateStudent.setMiddleName(request.getParameter("middle"));
    updateStudent.setLastName(request.getParameter("last"));
}
```

Delete Code

```java
// Check if a delete is requested
if (action != null && action.equals("delete")) {
    // remove the student at the given id
    students.remove(Integer.parseInt(request.getParameter("id")));
}
```
Table Presentation Code

```html
<table>
  <tr>
    <th>ID</th>
    <th>PID</th>
    <th>First Name</th>
    <th>Middle Name</th>
    <th>Last Name</th>
  </tr>
  < Insert Form Code >
  <% // Iterate through student map
    Iterator it = students.entrySet().iterator;
    while(it.hasNext()) {
      <Iteration Code>
      %>
    </table>
```

Insert Form Code

```html
<form action="students.jsp" method="POST">
  <input type="hidden" name="action" value="insert"/>
  <input type="hidden" name="pid" size="10"/>
  <input type="hidden" name="first" size="15"/>
  <input type="hidden" name="middle" size="15"/>
  <input type="hidden" name="last" size="15"/>
  <input type="submit" value="Insert"/>
</form>
```

Iteration Code

```html
<% Map.Entry pair = (Map.Entry)it.next(); %>
<form action="students.jsp" method="POST">
  <input type="hidden" name="action" value="update" />
  <input type="hidden" name="id" value="
    <%=(Student)pair.getValue()).getPID()%>
  />
  ...<br>
  <td>
    <input type="submit" value="Update"/>
  </td>
</form>
```
Comments

• A winner when business logic is simple and rapid development matters

• Model 1 breaks (or at least stops being simple) when the next page is decided dynamically from the state of the application
  – Eg, if the user chose pens continue with orderpens.jsp; but if the user chooses notebooks continue to ordernotebooks.jsp

• It can easily become messy because business logic and database programming are part of the jsp
  – Contrast with Model-View-Controller programming: More lines of code but more modular
  – Then use Spring, Struts MVC frameworks
  – Besides Java/JSPs, Ruby-on-Rails and many other language/framework combos follow MVC

What is Wrong with JSPs?

• Business logic & html content (presentation) mixed together

• Especially hard to maintain/evolve a program

• Still not very clean separation of web designer and web developer tasks