Web Application Development

Introduction
Because of the wide spread use of internet, web based applications are becoming vital part of IT infrastructure of large organizations. For example web based employee performance management systems are used by organizations for weekly or monthly reviews of employees. On the other hand online course registration and examination systems can allow students to study while staying at their homes.

Web Applications
In general a web application is a piece of code running at the server which facilitates a remote user connected to web server through HTTP protocol. HTTP protocol follows stateless Request-Response communication model. Client (usually a web-browser) sends

![Diagram of HTTP communication model](image)

Figure 1: A typical web application
A web server is software which provides users, access to the services that are present on the internet. These servers can provide support for many protocols used over internet or intranet like HTTP, FTP, telnet etc

HTTP Basics
A protocol defines the method and way of communication between two parties. For example when we talk to our teacher we use a certain way which is different from the way that we adopt with our friends or parents. Similarly there are many different protocols used by computers to communicate with each other depending on applications.

![Diagram of HTTP communication model](image)

Figure 2: HTTP communication model
-HTTP is as request-response oriented protocol.
-It is a stateless protocol since there is no built-in state management between successive requests. We will discuss state later.
Parts of an HTTP request

- Request Method: It tells the server the type of action that a client wants to perform
- URI: Uniform Resource Indictor specifies the address of required document or resource
- Header Fields: Optional headers can be used by client to tell server extra

```
GET /index.html HTTP/1.1 request line
Host: java.sun.com request headers
User-Agent: Mozilla/4.5 [en]
Accept: image/gif, image/jpeg, image/pjpeg, */*
Accept-Language: en
Accept-Charset: iso-8859-1,*,utf-8
```

Figure 3: HTTP request example

- Body: Contains data sent by client to the server
  - Other request headers like FROM (email of the person responsible for request) and VIA (used by gateways and proxies to show intermediate sites the request passes) can also be used.

Request Parameters

- Request can also contain addition information in form of request parameters
  1. In URL as query string e.g. http://www.gmail.com/register?name=ali&state=punjab
  2. As part of request body (see Figure 3)

Parts of HTTP response

Result Code: A numeric status code and its description. - Header Fields:
Servers use these fields to tell client about server information like configurations and software etc. - Body: Data sent by server as part of response that is finally seen by the user.
HTTP Response Codes

- An HTTP Response code tells the client about the state of the response i.e. whether it's a valid response or some error has occurred etc. HTTP Response codes fall into five general categories.

  - **100-199**
    Codes in the 100s are informational, indicating that the client should respond with some other action.
    100: Continue with partial request.

  - **200-299**
    Values in the 200s signify that the request was successful.
    200: Means everything is fine.

  - **300-399**
    Values in the 300s are used for files that have moved and usually include a Location header indicating the new address.
    300: Document requested can be found several places; they'll be listed in the returned document.

  - **400-499**
    Values in the 400s indicate an error by the client.
    404: Indicates that the requested resource is not available.
    401: Indicates that the request requires HTTP authentication.
    403: Indicates that access to the requested resource has been denied.

  - **500-599**
    Codes in the 500s signify an error by the server.
    503: Indicates that the HTTP server is temporarily overloaded and unable to handle the request.
• **404**: indicates that the requested resource is not available

![HTTP Status 404](image1)

• **401**: Indicates that request requires HTTP authentication

![HTTP Status 401](image2)

**Server Side Programming**

Web server pages can be either static pages or dynamic pages. A static web page is a simple HTML (Hyper Text Transfer Language) file. When a client requests an HTML page the server simple sends back response with the required page.
Figure 5: Static web page request and response

An example of static web page is given below. While in case of dynamic web pages server executes an application which generates HTML web pages according to specific requests coming from client. These dynamically generated web pages are sent back to client with the response.
Figure 6: Dynamic web page request and response
Why build Pages Dynamically?

We need to create dynamic web pages when the content of site changes frequently and client specific response is required. Some of the scenarios are listed below:

- The web page is based on data submitted by the user. e.g. results page from search engines and order confirmation pages at online stores.

The Web page is derived from data that changes frequently. e.g. a weather report or news headlines page.
The Web page uses information from databases or other server-side resources. e.g. an e-commerce site could use a servlet to build a Web page that lists the current price and availability of each item that is for sale.

- Using technologies for developing web pages that include dynamic content.
- Developing web based applications which can produce web pages that contain information that is connection-dependent or time-dependent.

**Dynamic Web Content Technologies Evolution**

Dynamic web content development technologies have evolved through time in speed, security, ease of use and complexity. Initially C based CGI programs were on the server. Then template based technologies like ASP and PHP were then introduced which allowed ease of use for designing complex web pages. Sun Java introduced Servlets and JSP that provided more speed and security as well as better tools for web page creation.
Figure 7: Dynamic web content technologies evolution
Layers and Web Application

Normally web applications are partitioned into logical layers. Each layer performs a specific functionality which should not be mixed with other layers. Layers are isolated from each other to reduce coupling between them but they provide interfaces to communicate with each other.

- **Presentation Layer:**
  Provides a user interface for client to interact with application. This is the only part of application visible to client.

- **Business Layer**
  The business or service layer implements the actual business logic or functionality of the application. For example in case of online shopping systems this layer handles transaction management.

- **Data Layer**
  This layer consists of objects that represent real-world business objects such as an Order, OrderLineItem, Product, and so on.
  There are several Java technologies available for web application development which includes Java Servlets, JavaServer Pages, JavaServer Faces etc.

References:
- Java, A Practical Guide by Umair Javed
- Java tutorial by Sun: http://java.sun.com/docs/books/tutorial/
Java Servlets

Servlets are a Java technology’s answer to CGI programming. CGI was widely used for generating dynamic content before Servlets arrived. They were programs written mostly in C, C++ that run on a web server and used to build web pages.

As you can see in the figure below, a client sends a request to the web server, the server forwards that request to a servlet, the servlet generates dynamic content, mostly in the form of HTML pages, and then returns it back to the server, which sends it back to the client. Hence, we can say that servlets are extending the functionality of the webserver (the job of the earlier servers was to respond only to request, by may be sending the required html file back to the client, and generally no processing was performed on the server).

What Servlets can do?
- Servlets can do anything that a java class can do. For example, connecting with database, reading/writing data to/from file etc.
- Handles requests sent by the user (clients) and generates response dynamically (normally HTML pages).
- The dynamically generated content is sent back to the user through a webserver (client)

Servlets vs. other SSP Technologies
The java’s servlet technology has following advantage over their counter parts:

Convenient
Servlets can use the whole Java API e.g. JDBC. So if you already know Java, why learn Perl or C. Servlets have an extensive infrastructure for automatically parsing and decoding HTML form data, reading and sending HTTP headers, handling cookies and tracking session etc and many more utilities

Efficient
With traditional CGI, a new process is started for each request while with servlets each request is handled by a lightweight java thread, not a heavy weight operating system process. (more on this later)

Powerful
Java servlets let you easily do several things that are difficult or impossible with regular CGI. For example, servlets can also share data among each other
Portable
Since Java is portable and servlets is a Java-based technology therefore they are generally portable across web servers.

Inexpensive
There are numbers of free or inexpensive web servers available that are good for personal use or low volume web sites. For example Apache is a commercial grade webserver that is absolutely free. However some very high end web and application servers are quite expensive e.g BEA weblogic. We’ll also use Apache in this course.

Software Requirements
To use Java servlets will be needed:
- J2SE
- Additional J2EE based libraries for servlets such as servlet-api.jar and jsp-api.jar. Since these libraries are not part of J2SE, you can download these APIs separately. However these APIs are also available with the web server you’ll be using.
- A capable servlet web engine (webserver)

Jakarta Servlet Engine (Tomcat)
Jakarta is an Apache project and tomcat is a projects. Apache Tomcat is an open source web server, which is used as an official reference implementation of Java Servlets and Java Server Pages technologies.
Tomcat is developed in an open and participatory environment and released under the Apache software license.

Environment Setup
To work with servlets and JSP technologies, you first need to set up the environment. Tomcat installation can be performed in two different ways (a) using .zip file (b) using .exe file. This setup process is broken down into the following steps:

1. Download the Apache Tomcat Server
2. Install Tomcat
3. Set the JAVA_HOME variable
4. Set the CATALINA_HOME variable
5. Set the CLASSPATH variable
6. Test the Server

Environment Setup Using .zip File
Let’s take a detail look on each step and get some hands on experience of environment setup.

1. Download the Apache Tomcat Server
From the [http://tomcat.apache.org](http://tomcat.apache.org), download the zip file for the current release (e.g. jakarta-tomcat-5.5.9.zip or any latest version) on your C:\ drive. There are different releases available on site. Select to download .zip file from the Binary Distributions /E core section.

Note: J2SE 5.0 must be installed prior to use the 5.5.9 version of tomcat.
2. Installing Tomcat using .zip file
- Unzip the file into a location (e.g. C:\). (Rightclick on the zip file and select unzipped option.)
- When the zip file will unzipped a directory structure will be created on your computer such as:

```
Local Disk (C:)
  Documents and Settings
  downloads
  Java
  Inetpub
  jakarta-tomcat-5.5.9
  bin
```

- The C:\jakarta-tomcat-5.5.9 folder is generally referred as root directory or CATALINA_HOME

**Note:** After extraction, make sure C:\jakarta-tomcat-5.5.9 contains a bin subdirectory. Sometimes students create their own directory and unzip the file there such as C:\jakarta-tomcat-5.5.9\jakarta-tomcat-5.5.9. This causes problems while giving path information

3. Set the JAVA_HOME variable

- JAVA_HOME indicates the root directory of your jdk. Set the JAVA_HOME environment variable to tell Tomcat, where to find java
- This variable should list the base JDK installation directory, not the bin subdirectory
- To set it, right click on My Computer icon. Select the advanced tab, a System Properties window will appear in front of you like shown below. Select the Environment Variables button to proceed.
- On clicking Environment Variable button, the Environment Variables window will open as shown next
- Create a new User variable by clicking New button as shown above, the New User Variable window will appear
- Set name of variable JAVA_HOME
- The value is the installation directory of JDK (for example C:\ProgramFiles\j2sdk_nb\j2sdk1.4.2). This is shown below in the picture. **Please note that bin folder is not included in the path.**
- Press Ok button to finish
4. **Set the CATALINA_HOME variable**

CATALINA_HOME is used to tell the system about the root directory of the TOMCAT. There are various files (classes, exe etc) needed by the system to run. CATALINA_HOME is used to tell your system (in this case your web server Tomcat) where the required files are.

- To Set the CATALINA_HOME environment variable, create another User Variable.
- Type CATALINA_HOME as the name of the environment variable.
- Its value should be the path till your top-level Tomcat directory. If you have unzipped the Tomcat in C drive. It should be C:\jakarta-tomcat-5.5.9. This is shown below:
- Press Ok button to finish

![New User Variable](image)

**Note:** To run Tomcat (web server) you need to set only the two environment variables and these are JAVA_HOME & CATALINA_HOME

5. **Set the CLASSPATH variable**

Since servlets and JSP are not part of the Java 2 platform, standard edition, you have to identify the servlet classes to the compiler. The server already knows about the servlet classes, but the compiler (i.e., javac) you use for compiling source files of servlet does not. So if you don't set your CLASSPATH, any attempt to compile servlets, tag libraries, or other classes that use the servlet API will fail with error messages about unknown classes.

- To Set the CLASSPATH environment variable, create another User Variable.
- Type CLASSPATH as the name of the environment variable.
- Its value should be the path for servlet-api.jar and jsp-api.jar. These file can be found on following path:
  - C:\jakarta-tomcat-5.5.9\common\lib\**[javax-api.jar]**
  - C:\jakarta-tomcat-5.5.9\common\lib\**[jsp-api.jar]**
  - Both these api's are specified as values with semicolon between them. Remember to add semicolon dot semicolon (.;) at the end too. For example
Classpath = C:\jakarta-tomcat-5.5.9\common\lib\servlet-api.jar;C:\jakarta-tomcat-5.5.9\common\lib\jsp-api.jar;
This is also shown in the figure below

<table>
<thead>
<tr>
<th>New User Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable Name:</td>
</tr>
<tr>
<td>Variable Value:</td>
</tr>
</tbody>
</table>

— Press OK button to finish the setting of CLASSPATH variable

6. Test the server
Before making your own servlets and JSP, verify that the server is working properly. Follow these steps in order to do that:

• Open the C:\jakarta-tomcat-5.5.9\bin folder and locate the startup.bat file.

• Double clicking on this file will open up a DOS window, which will disappear, and another DOS window will appear, the second window will stay there. If it does not your paths are not correctly set.

• Now to check whether your server is working or not, open up a browser window and type http://localhost:8080. This should open the default page of Tomcat as shown in next diagram:

Note: If default page doesn’t displayed, open up an internet explorer window, move on to Tools Æ Internet Options Æ Connections LAN Settings. Make sure that option of “Bypass proxy server for local addresses” is unchecked.
There is another easier way to carry out the environment setup using .exe file. However, it is strongly recommended that you must complete the environment setup using .zip file to know the essential fundamentals.

Environment Setup Using .exe File
Let’s look at the steps involved to accomplish the environment setup using .exe file.

1. Download the Apache Tomcat Server

From the [http://tomcat.apache.org](http://tomcat.apache.org), download the .exe file for the current release (e.g. jakarta-tomcat-5.5.9.zip) on your C:\ drive. There are different releases available on site. Select to download Windows executable (.exe) file from Binary Distributions Core section.

**Note:** J2SE 5.0 must be installed to use the 5.5.9 version of tomcat.

2. Installing Tomcat using .exe file
   - Run the .exe file by double clicking on it.
   - Moving forward in setup, you will reach to the following window - Select install type “Full” and press Next button to proceed.
Choose the folder in which you want to install Apache Tomcat and press Next to proceed.

The configuration window will be opened. Leave the port unchanged (since by default web servers run on port 8080, you can change it if you really want to). Specify the user name & password in the specified fields and press Next button to move forward. This is also shown in the diagram coming next:
Apache Tomcat Setup: Configuration Options

**Configuration**
Tomcat basic configuration.

- **HTTP/1.1 Connector Port**: 8080
- **Administrator Login**
  - **User Name**: umair
  - **Password**: •••••

Nullsoft Install System v2.0
-The setup will automatically select the Java Virtual Machine path. Click Install button to move ahead.
-Finish the setup with the Run Apache Tomcat option selected. It will cause the tomcat server to run in quick launch bar as shown in diagram below. The Apache Tomcat shortcuts will also added to Programs menu.

-Double clicking on this button will open up Apache Tomcat Properties window. From here you can start or stop your web server. You can also configure many options if you want to. This properties window is shown below:

3. Set the JAVA_HOME variable
Choosing .exe mode does not require completing this step.

4. Set the CATALINA_HOME variable
Choosing .exe mode does not require completing this step.

5. Set the CLASSPATH variable
Same as step 5 of .zip installation mode

6. Test the server
If tomcat installation is made using .exe file, follow these steps
- Open the Apache Tomcat properties window by clicking on the Apache Tomcat button from Quick Launch.
- Start the tomcat server if it is not running by clicking on Start button.
- Open up a browser window and type http://localhost:8080. This should open the default page of Tomcat as shown in the next diagram:

**Note:** If default page doesn’t displayed, open up an internet explorer window, move on to Tools → Internet Options → Connections LAN Settings. Make sure that option of “Bypass proxy server for local addresses” is unchecked.

![Tomcat Default Page](image)

**References:**
- Java, A Lab Course by Umair Javed
- Java Servlet & JSP tutorial [http://www apl jhu edu/~hall/java/Servlet Tutorial/](http://www apl jhu edu/~hall/java/Servlet Tutorial/)
Creating a Simple Web Application in Tomcat

In this handout, we’ll discuss the standard tomcat directory structure, a pre-requisite for building any web application. Different nuts and bolts of Servlets will also be discussed. In the later part of this handout, we’ll also learn how to make a simple web application using servlet.

**Standard Directory Structure of a J2EE Web Application**

A web application is defined as a hierarchy of directories and files in a standard layout. Such hierarchies can be used in two forms

- **Unpack**
  - Where each directory & file exists in the file system separately
  - Used mostly during development

- **Pack**
  - Known as Web Archive (WAR) file
  - Mostly used to deploy web applications

The webapps folder is the top-level Tomcat directory that contains all the web applications deployed on the server. Each application is deployed in a separate folder often referred as “context”.

To make a new application e.g. myapp in tomcat you need a specific folder hierarchy.

- Create a folder named myapp in C:\jakarta-tomcat-5.5.9\webapps folder. This name will also appear in the URL for your application. For example

- All JSP and html files will be kept in main application folder (C:\jakartatomecat-5.5.9\webapps\myapp)
- Create another folder inside myapp folder and change its name to WEB-INF. Remember WEB-INF is case sensitive and it is **not** WEB-INF
Configuration files such as web.xml will go in WEB-INF folder (C:\jakarta-tomcat-5.5.9\webapps\myapp\WEB-INF)

Create another folder inside WEB-INF folder and change its name to classes. Remember classes name is also case sensitive.

Servlets and Java Beans will go in classes folder (C:\jakarta-tomcat5.5.9\webapps\myapp\WEB-INF\classes)

That’s the minimum directory structure required in order to get started. This is also shown in the figure below:

-To test application hierarchy, make a simple html file e.g. index.html file. Write some basic HTML code into it and save it in main application directory i.e.
C:\jakarta-tomcat-5.5.9\webapps\myapp\index.html

-Restart the server and access it by using the URL http://localhost:8080/myapp/index.html

-A more detailed view of the Tomcat standard directory structure is given below.
Here you can see some other folders like lib& tags under the WEB-INF.
The lib folder is required if you want to use some achieve files (.jar). For example an API in jar format that can help generating .pdf files.
-Similarly tags folder is helpful for building custom tags or for using .tagfiles.

**Note:** Restart Tomcat every time you create a new directory structure, a servlet or a java bean so that it can recognize it. For JSP and html files you don’t have to restart the server.

### Writing Servlets

#### Servlet Types

-Servlet related classes are included in two main packages javax.servlet and javax.servlet.http.
-Every servlet must implement the javax.servlet.Servlet interface, it contains the servlet’s life cycle methods etc. ([Life cycle methods will be discussed in next handout](https://example.com/life-cycle-methods)).
-In order to write your own servlet, you can subclass from GenericServlet or HttpServlet

#### GenericServlet class

- Available in javax.servlet package
- Implements javax.servlet.Servlet
- Extend your class from this class if you are interested in writing protocol independent servlets

#### HttpServlet class

- Available in javax.servlet.http package
- Extends from GenericServlet
- Adds functionality for writing HTTP specific servlets as compared to GenericServlet
- Extend your class from HttpServlet, if you want to write HTTP based servlets

**Servlet Class Hierarchy**

The Servlet class hierarchy is given below. Like all java classes GenericServlet also inherits from Object class. Apart from GenericServlet and HttpServlet classes, ServletRequest, HttpServletRequest, ServletResponse and HttpServletResponse are also helpful in writing a servlet. As you can guess ServletRequest & ServletResponse are used in conjunction with GenericServlet. These classes are used for processing protocol independent requests and generating protocol independent responses respectively.

![Servlet Class Hierarchy Diagram]

HttpServletRequest & HttpServletResponse are used for processing HTTP protocol specific requests and generating HTTP specific response. Obviously these classes will be used in conjunction with HttpServlet class, which means you are making a HTTP protocol specific servlet.

**Types of HTTP requests**

HTTP supports different types of request to be sent over to server. Each request has some specific purpose. The most important ones are **get** & **post**. Given below a brief overview of each request type is given. You can refer to RFC of HTTP for further details.

**-GET:** Requests a page from the server. This is the normal request used when browsing web pages.

**-POST:** This request is used to pass information to the server. Its most common use is with HTML forms.

**-PUT:** Used to put a new web page on a server.

**-DELETE:** Used to delete a web page from the server.

**-OPTIONS:** Intended for use with the web server, listing the supported options.
**-TRACE:** Used to trace servers

**GET & POST, HTTP request types**

Some details on GET and POST HTTP request types are given below.

**. GET**

- Attribute-Value pair is attached with requested URL after '?'.
  - For example if attribute is ‘name’ and value is ‘ali’ then the request will be http://www.gmail.com/register?name=ali
  - For HTTP based servlet, override doGet() methods of HttpServlet class to handle these type of requests.

**. POST**

- Attribute-Value pair attached within the request body. For your reference HTTP request diagram is given below again:

```plaintext
GET /index.html HTTP/1.1 request line

Host: java.sun.com request headers
User-Agent: Mozilla/4.5 [en]
Accept: image/gif, image/jpeg, image/pjpeg, */*
Accept-Language: en
Accept-Charset: iso-8859-1,*,utf-8

optional request body
```

- Override doPost() method of HttpServlet class to handle POST type requests.

**Steps for making a Hello World Servlet**

To get started we will make a customary “HelloWorldServlet”. Let’s see what are the steps involved in writing a servlet that will produce “Hello World”

1. Create a directory structure for your application (i.e. helloapp). This is a one time process for any application
2. Create a HelloWorldServlet source file by extending this class from HttpServlet and overriding your desired method. For example doGet() or doPost().
3. Compile it (If get error of not having required packages, check your class path)
4. Place the class file of HelloWorldServlet in the classes folder of your web application (i.e. myapp).

**Note:** If you are using packages then create a complete structure under classes folder

5. Create a deployment descriptor (web.xml) and put it inside WEB-INF folder
6. Restart your server if already running
7. Access it using Web browser
Example Code: HelloWorldServlet.java

//File HelloWorldServlet.java

// importing required packages
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

// extending class from HttpServlet
public class HelloWorldServlet extends HttpServlet {

    /* overriding doGet() method because writing a URL in the browser by default generates a request of
    GET type
    As you can see, HttpServletRequest and HttpServletResponse are passed to this
    method. These objects will help in processing of HTTP request and generating
    response for HTTP
    This method can throw ServletException or IOException, so we mention these exception
    types after method signature */
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        /* getting output stream i.e PrintWriter from response object by calling getWriter
        method on it
        As mentioned, for generating response, we will
        use HttpServletResponse object*/
        PrintWriter out = response.getWriter();

        /* printing Hello World in the browser using PrintWriter
        object. You can also write HTML like
        out.println("<h1> Hello World </h1>") */
        out.println("Hello World!");
    } // end doGet()
} // end HelloWorldServlet

Example Code: web.xml
eXtensible Markup Language (xml) contains custom defined tags which convey information about the
content. To learn more about XML visit http://www.w3schools.com.
Inside web.xml, the <web-app> is the root tag representing the web application. All other tags come inside
of it.

<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app>
    <servlet> <servlet-name> HelloWorldServlet </servlet-name> <servlet-class>
        HelloWorldServlet </servlet-class>
    </servlet>
</web-app>
<servlet-mapping>
<servlet-name> HelloWorldServlet </servlet-name>
</servlet-mapping>
The `<servlet>` tag represents one's servlet name and its class. To specify the name of servlet, `<servlet-name>` tag is used. Similarly to specify the class name of servlet (it is the same name you used for making a servlet), `<servlet-class>` tag is used.

**Note:** It is important to note here that you can specify any name for a servlet inside `<servlet-name>` tag. This name is used for referring to servlet in later part of web.xml. You can think of it as your id assigned to you by your university while you have actually different name (like `<servlet-class>`).

Next we will define the servlet mapping. By defining servlet mapping we are specifying URL to access a servlet. `<servlet-mapping>` tag is used for this purpose.

Inside `<servlet-mapping>` tag, first you will write the name of the servlet for which you want to specify the URL mapping using `<servlet-name>` tag and then you will define the URL pattern using `<url-pattern>` tag. Notice the forward slash (/) is used before specifying the url. You can specify any name of URL. The forward slash indicates the root of your application.

```xml
<servlet-mapping>
  <servlet-name>myfirstservlet</servlet-name>
  <url-pattern>/myfirstservlet</url-pattern>
</servlet-mapping>
```

Now you can access HelloWorldServlet (if it is placed in myapp application) by giving the following url in the browser

```
http://localhost:8080/myapp/myfirstservlet
```

**Note:** Save this web.xml file by placing double quotes ("web.xml") around it as you did to save .java files.

### Compiling and Invoking Servlets

- Compile HelloWorldServlet.java using javac command.
- Put HelloWorldServlet.class in C:\jakarta-tomcat5.5.9\webapps\myapp\WEB-INF\classes folder
- Put web.xml file in C:\jakarta-tomcat5.5.9\webapps\myapp\WEB-INF folder
- Invoke your servlet by writing following URL in web browser. Don’t forget to restart your tomcat server if already running

```
http://localhost:8080/myapp/myfirstservlet
```

**Note:** By using IDEs like netBeans® 4.1, you don’t have to write web.xml by yourself or even to worry about creating directory structure and to copy files in appropriate locations. However manually undergoing this process will strengthen your concepts and will help you to understand the underlying mechanics ☺.

### References:

Entire material for this handout is taken from the book **JAVA A Lab Course** by **Umair Javed**. This material is available just for the use of VU students of the course Web Design and Development and not for any other commercial purpose without the consent of author.
Servlets Lifecycle

In the last handout, we have seen how to write a simple servlet. In this handout we will look more specifically on how servlets get created and destroyed. What different set of method are invoked during the lifecycle of a typical servlet. The second part consists on reading HTML form data through servlet technology. This will be explored in detail using code example.

Stages of Servlet Lifecycle

A servlet passes through the following stages in its life.
1. Initialize
2. Service
3. Destroy

As you can conclude from the diagram below, that with the passage of time a servlet passes through these stages one after another.

1. Initialize

When the servlet is first created, it is in the initialization stage. The webserver invokes the init() method of the servlet in this stage. It should be noted here that init() is only called once and is not called for each request. Since there is no constructor available in Servlet so this urges its use for one time initialization (loading of resources, setting of parameters etc) just as the init() method of applet.

Initialize stage has the following characteristics and usage
- Executed once, when the servlet gets loaded for the first time
- Not called for each client request
- The above two points make it an ideal place to perform the startup tasks which are done in constructor in a normal class.

2. Service

The service() method is the engine of the servlet, which actually processes the client’s request. On every request from the client, the server spawns a new thread and calls the service() method as shown in the figure below. This makes it more efficient as compared to the technologies that use single thread to respond to requests.

© Copyright Virtual University of Pakistan
The figure below show both versions of the implementation of service cycle. In the upper part of diagram, we assume that servlet is made by sub-classing from GenericServlet. (Remember, GenericServlet is used for constructing protocol independent servlets.) To provide the desired functionality, service() method is overridden. The client sends a request to the web server; a new thread is created to serve this request followed by calling the service() method. Finally a response is prepared and sent back to the user according to the request.

The second part of the figure illustrates a situation in which servlet is made using HttpServlet class. Now, this servlet can only serves the HTTP type requests. In these servlets doGet() and doPost() are overridden to provide desired behaviors. When a request is sent to the web server, the web server after creating a thread, passes on this request to service() method. The service() method checks the HTTP requests type (GET, POST etc) and calls the doGet() or doPost() method depending on how the request is originally sent. After forming the response by doGet() or doPost() method, the response is sent back to the service() method that is finally sent to the user by the web server.
3. Destroy

The web server may decide to remove a previously loaded servlet instance, perhaps because it is explicitly asked to do so by the server administrator, or perhaps servlet container shuts down or the servlet is idle for a long time, or may be the server is overloaded. Before it does, however it calls the servlets destroy() method. This makes it a perfect spot for releasing the acquired resources.

Summary

- A Servlet is constructed and initialized. The initialization can be performed inside of init() method.
- Servlet services zero or more requests by calling service() method that may decide to call further methods depending upon the Servlet type (Generic or HTTP specific)
- Server shuts down, Servlet is destroyed and garbage is collected

The following figure can help to summarize the life cycle of the Servlet

![Servlet Life Cycle Diagram](image)

The web server creates a servlet instance. After successful creation, the servlet enters into initialization phase. Here, init() method is invoked for once. In case web server fails in previous two stages, the servlet instance is unloaded from the server.

After initialization stage, the Servlet becomes available to serve the clients requests and to generate response accordingly. Finally, the servlet is destroyed and unloaded from web server.

**Reading HTML Form Data Using Servlets**

In the second part, the required concepts and servlet technology is explored in order to read HTML form data. To begin with, let’s first identify in how many ways a client can send data
HTML & Servlets
Generally HTML is used as a Graphics User Interface for a Servlet. In the figure below, HTML form is being used as a GUI interface for MyServlet. The data entered by the user in HTML form is transmitted to the MyServlet that can process this data once it read out. Response may be generated to fulfill the application requirements.

Types of Data send to Web Server
When a user submits a browser request to a web server, it sends two categories of data:

. Form Data
   Data, that the user explicitly type into an HTML form. For example: registration information provided for creating a new email account.

. HTTP Request Header Data
   Data, which is automatically, appended to the HTTP Request from the client for example, cookies, browser type, and browser IP address.

Based on our understanding of HTML, we now know how to create user forms. We also know how to gather user data via all the form controls: text, password, select, checkbox, radio buttons, etc. Now, the question arises: if I submit form data to a Servlet, how do I extract this form data from servlet? Figuring this out, provides the basis for creating interactive web applications that respond to user requests.

Reading HTML Form Data from Servlet
Now let see how we can read data from “HTML form” using Servlet. The HttpServletRequest object contains three main methods for extracting form data submitted by the user:

. getParameter(String name)
   -Used to retrieve a single form parameter and returns String corresponding to name specified.
   -Empty String is returned in the case when user does not enter any thing in the specified form field.
   -If the name specified to retrieve the value does not exist, it returns null.
   Note: You should only use this method when you are sure that the parameter has only one value. If the parameter might have more than one value, use getParameterValues().
.getParameterValues(String name)
   -Returns an array of Strings objects containing all of the given values of the given request parameter.
   -If the name specified does not exist, null is returned

.getParameterNames() 
   -If you are unsure about the parameter names, this method will be helpful
   -It returns Enumeration of String objects containing the names of the parameters that come with the request.
   -If the request has no parameters, the method returns an empty Enumeration.

Note: All these methods discussed above work the same way regardless of the request type(GET or POST). Also remember that form elements are case sensitive for example, “userName” is not the same as the “username.”

Example Code: Reading Form Data using Servlet
This example consists of one HTML page (index.html), one servlet (MyServlet.java) and one xml file (web.xml) file. The HTML page contains two form parameters: firstName and surName. The Servlet extracts these specific parameters and echoes them back to the browser after appending “Hello”.

Note: The example given below and examples later in coming handouts are built using netBeans®4.1. It's important to note that tomcat server bundled with netBeans® runs on 8084 port by default.

index.html
Let's discuss the code of above HTML form. As you can see in the <FORM> tag, the attribute METHOD is set to “GET”. The possible values for this attribute can be GET and POST. Now what do these values mean?

- Setting the method attribue to “GET” means that we want to send the HTTP request using the GET method which will eventually activate the doGet() method of the servlet. In the GET method the information in the input fields entered by the user, merges with the URL as the query string and are visible to the user.
- Setting METHOD value to “POST” hides the entered information from the user as this information becomes the part of request body and activates doPost() method of the servlet.

Attribute ACTION of <FORM> tag is set to http://localhost:8084/paramapp.formservlet. The form data will be transmitted to this URL. paramapp is the name of web application created using netBeans. formservlet is the value of <url-pattern> defined in the web.xml. The code of web.xml is given at the end. The NAME attribute is set to “myform” that helps when the same page has more than one forms. However, here it is used only for demonstration purpose.

To create the text fields where user can enter data, following lines of code come into play

```html
<INPUT TYPE = “text” NAME = "firstName">
<INPUT TYPE = “text” NAME = "surName">
```

Each text field is distinguished on the basis of name assigned to them. Later these names also help in
extracting the values entered into these text fields.
MyServlet.java

Now lets take a look at the servlet code to which HTML form data is submitted.
import java.io.*; import javax.servlet.*; import
javax.servlet.http.*;
public class MyServlet extends HttpServlet{
    public void doGet(HttpServletRequest req,            HttpServletResponse res) throws ServletException, IOException{
        // reading first name parameter/textfield
        String fName = req.getParameter("firstName");

        // reading surname parameter/textfield
        String sName = req.getParameter("surName");
        // getting stream from HttpServletRequest object
        PrintWriter out = res.getWriter();
        out.println("Hello: " + fName + " " + sName);
        out.close();
    }
} // end FormServlet

We started the code with importing three packages.

    import java.io.*;
    import javax.servlet.*;
    import javax.servlet.http.*;

These packages are imported to have the access on PrintWriter, HttpServletRequest, HttpServletResponse, ServletException and IOException classes.

The class MyServlet extends from HttpServlet to inherit the HTTP specific functionality. If you recall HTML code (index.html) discussed above, the value of method attribute was set to “GET”. So in this case, we only need to override doGet() method.

Entering inside doGet() method brings the crux of the code. These are:

    String fName = req.getParameter("firstName");
    String sName = req.getParameter("surName");

Two String variables fName and sName are declared that receive String values returned by getParameter() method. As discussed earlier, this method returns String corresponds to the form parameter. Note that the values of name attributes of input tags used in index.html have same case with the ones passed to getParameter() methods as parameters. The part of HTML code is reproduced over here again:

    <INPUT TYPE = “text” NAME="firstName">
    <INPUT TYPE = “text” NAME="surName">
In the last part of the code, we get the object of PrintWriter stream from the object of
HttpServletResponse. This object will be used to send data back the response. Using PrintWriter object (out), the names are printed with appended “Hello” that becomes visible in the browser.

web.xml

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>

<web-app>
  <servlet>
  <servlet-name>FormServlet</servlet-name>
  <servlet-class>MyServlet</servlet-class>
  </servlet>
  <servlet-mapping>
  <servlet-name>FormServlet</servlet-name>
  <url-pattern>form servlet</url-pattern>
  </servlet-mapping>
</web-app>
```

The `<servlet-mapping>` tag contains two tags `<servlet-name>` and `<url-pattern>` containing name and pattern of the URL respectively. Recall the value of action attribute of the `<form>` element in the HTML page. You can see it is exactly the same as mentioned in `<url-pattern>` tag.

http://localhost:8084/paramapp/formservlet

**References:**
- JAVA a Lab Course by Umair Javed
- Java API documentation
- Core Servlets and JSP by Marty Hall
More on Servlets

The objective of this handout is to learn about the use and implementation of initialization parameters for a Servlet. Moreover different ways of redirecting response and forwarding or including requests also discussed in detail.

Initialization Parameters

Some times at the time of starting up the application we need to provide some initial information e.g. name of the file where server can store logging information, DSN for database etc. Initial configuration can be defined for a Servlet by defining some string parameters in web.xml. This allows a Servlet to have initial parameters from outside. This is similar to providing command line parameters to a standard console based application.

Example: setting init parameters in web.xml
Let’s have a look on the way of defining these parameters in web.xml

```xml
<init-param> //defining param 1
    <param-name> param1 </param-name>
    <param-value> value1 </param-value>
</init-param>
<init-param> //defining param 2
    <param-name> param2 </param-name>
    <param-value> value2 </param-value>
```

In the above code, it is shown that for each parameter we need to define separate <initparam> tag that have two sub tags <param-name> and <param-value>, which contain the name and values of the parameter respectively.

ServletConfig

Every Servlet has an object called ServletConfig associated with it as shown in the fig. below. It contains relevant information about the Servlet like initialization parameters defined in web.xml
Reading Initialization Parameters
Now let’s see, how we can access init parameters inside the Servlet. The method getInitParameter() of ServletConfig is usually used to access init parameters. It takes a String as a parameter, matches it with <param-name> tag under all <init-param> tags and returns <param-value> from the web.xml
One way is to override init() method as shown in the code below. The ServletConfig object can then be used to read initialization parameter.

```java
public void init(ServletConfig config) throws ServletException {
    String name = config.getInitParameter("paramName");
}
```

Another way to read initialization parameters outside the init() method is
- Call getServletConfig() to obtain the ServletConfig object
- Use getInitParameter() of ServletConfig to read initialization parameters

```java
public void anyMethod() { // defined inside servlet
    ServletConfig config = getServletConfig();
    String name = config.getInitParameter("param_name");
}
```

Example Code: Reading init parameters
MyServlet.java will read the init parameter (log file name) defined inside web.xml. The code is given below:

```java
import java.io.*;
import java.net.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class MyServlet extends HttpServlet {
    // attribute used to store init-parameter value
    String fileName;

    // overriding init() method
    public void init(ServletConfig config) throws ServletException {
        super.init(config);
        // reading init-parameter “logfilename” stored in web.xml
        fileName = config.getInitParameter("logfilename");
    }

    /* Both doGet() & doPost() methods are override over here, processRequest() is called from both these methods. This makes possible for a servlet to handle both POST and GET requests identically.
    */
    // Handles the HTTP GET request type
    protected void doGet(HttpServletRequest request,
    HttpServletResponse response) throws ServletException, IOException {
        processRequest(request, response);
    }
```
// Handles the HTTP POST request type
protected void doPost(HttpServletRequest request,
    HttpServletResponse response)throws ServletException, IOException {
    processRequest(request, response);
}
// called from doGet() & doPost()
protected void processRequest(HttpServletRequest request,
    HttpServletResponse response)throws ServletException, IOException {
    PrintWriter out = response.getWriter();
    // writing init-parameter value that is store in fileName
    out.println(fileName);
    out.close();
}

web.xml

<?xml version="1.0" encoding="UTF-8"?><web-app>
    <servlet> <servlet-name> MyServlet </servlet-name> <servlet-class> MyServlet </servlet-class>
        <init-param> <param-name> logfilename </param-name> <param-value> logoutput.txt </param-value>
        </init-param>
    </servlet>
    <servlet-mapping>
        <servlet-name> MyServlet </servlet-name>
        <url-pattern> /myservlet </url-pattern>
    </servlet-mapping>
</web-app>
Response Redirection

We can redirect the response of the Servlet to another application resource (another Servlet, an HTML page or a JSP) but the resource (URL) must be available to the calling Servlet, in the same Servlet context (discussed later).

There are two forms of response redirection that can be possible:

- Sending a standard redirect
- Sending a redirect to an error page

Sending a standard Redirect

- Using response.sendRedirect(“myHtml.html”) method, a new request is generated which redirects the user to the specified URL.
- If the URL is of another Servlet, that second Servlet will not have access to the original request object. For example, if the request is redirected from servlet1 to servlet2, then servlet2 would not be able to access the request object of servlet1.
- To have access to the original request object, you must use the request dispatching technique (discussed later) instead of redirect.

Sending a redirect to an error page

Instead of using response.sendRedirect(), we can use response.sendError() to show user an error page. This methods takes two parameters, first the error number that is a predefined constant of the response class (listed below) and second the appropriate error message. The steps to redirect the user to an error page are:

- An error code is sent as a parameter of response.sendError(int, msg) method
- The error page is displayed with the msg passed to method
- The error numbers are predefined constants of the HttpServletResponse class. For example:

  - SC_NOT_FOUND (404)
  - SC_NO_CONTENT (204)
  - SC_REQUEST_TIMEOUT (408)

Example Code: Response Redirection

The example given below demonstrates a typical sign on example in which a user is asked to provide login/password, providing correct information leads to welcome page or otherwise to a registration page. This example consists of login.html, welcome.html, register.html and MyServlet.java files. Let’s examine these one after another.

login.html

This page contains two text fields; one for entering username and another for password. The data from this page is submitted to MyServlet.java.
<html>
  <body>
    <h2>Please provide login details</h2>
    <form method="POST"
      action="http://localhost:8084/redirectionex/myservlet" name="myForm">
      User Id: <input type="text" name="userid" />
      Password: <input type="password" name="pwd" /> 
      <input type="submit" value="Submit Form" />
    </form>
  </body>
</html>

**welcome.html**

The user is directed to this page only if user provides correct login / password. This page only displays a successfully logged-in message to the user.

<html>
  <body>
    <h2>You have successfully logged in</h2>
  </body>
</html>

**register.html**

The user is redirected to this page in case of providing incorrect login/password information. The user can enter user id, address and phone number here to register.

Note: The code given below will only show fields to the user. It does not register user as no such functionality is added into this small example.

<html>
  <body>
    <h2>Your login is incorrect. Please register yourself</h2>
    <form method="POST"
      action="" name="myForm">
      Name: <input type="text" />
    </form>
  </body>
</html>
MyServlet.java

MyServlet.java accepts requests from login.html and redirects the user to welcome.html or register.html based on the verification of username & password provided. Username & password are compared with fix values in this example, however you can verify these from database or from a text file etc.

```java
import java.io.*;
import java.net.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class MyServlet extends HttpServlet {
    // Handles the HTTP GET request type
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        processRequest(request, response);
    }
    // Handles the HTTP POST request type
    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        processRequest(request, response);
    }
    protected void processRequest(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        String id = request.getParameter("userid");
        String pwd = request.getParameter("pwd");
        // comparing id & password with fix values
        if (id.equals("ali") &
            pwd.equals("vu")) {
            // redirecting user to welcome.html
            response.sendRedirect("welcome.html");
        }
    }
}
```
else { // redirecting user to register.html
response.sendRedirect("register.html");

/* if you want to display an error message to the user, you can use the following method */
response.sendError(response.SC_PROXY_AUTHENTICATION_REQUIRED,"Send Error Demo");*/

} // end else

}

ServletContext

ServletContext belongs to one web application. Therefore it can be used for sharing resources among
servlets in the same web application.

As initialization parameters, for a single servlet are stored in ServletConfig, ServletContext can store
initialization parameters for the entire web application. These parameters are also called context attributes
and exist for the lifetime of the application. The following figure illustrates the sharing of context attributes
among all the servlets of a web application.

![ServletContext Diagram]

Note:
- There is a single ServletContext per web application
- Different servlets will get the same ServletContext object, when calling getServletContext() during
different sessions

Request Dispatcher

RequestDispatcher provides a way to forward or include data from another source. The method
getRequestDispatcher(String path) of ServletContext returns a RequestDispatcher object associated with the
resource at the given path passed as a parameter.

Two important methods of RequestDispatcher are:
- `forward(ServletRequest req, ServletResponse resp)`
- `include(ServletRequest req, ServletResponse resp)`

**RequestDispatcher: forward**

Characteristics of forward methods are:
- It allows a Servlet to forward the request to another resource (Servlet, JSP or HTML file) in the same Servlet context.
- Forwarding remains transparent to the client unlike `res.sendRedirect(String location)`. You can not see the changes in the URL.
- Request Object is available to the called resource. In other words, it remains in scope.
- Before forwarding request to another source, headers or status codes can be set, but output content cannot be added.

To clarify the concepts, lets take the help from following figure. User initiates the request to servlet1. servlet1 forwards the request to servlet2 by calling `forward(request, response)`. Finally a response is returned back to the user by servlet2.

**RequestDispatcher: include**

It allows a Servlet to include the results of another resource in its response. The two major differences from forward are:
- Data can be written to the response before an include
- The first Servlet which receive the request, is the one which finishes the response

It will be more cleared from the following figure. User sends a `HTTPRequest` to Servlet1. Servlet2 is called by Servlet1 by using `include(request, response)` method. The response generated by Servlet2 sends back to
Servlet1. Servlet1 can also add its own response content and finally send it back to user.

References:
- Java A Lab Course by Umair Javed
- Core Servlets and JSP by Marty Hall
- Java API documentation
Dispatching Requests

In this handout we will start with request dispatching techniques and give some examples related to that. Further more some methods of HttpResponse and HttpRequest will also be discussed. Finally, this handout will be concluded by discussing the importance of session tracking. Before starting, let's take a look at the summery of the previous lecture.

Recap

In the previous lecture we had some discussion about Response Redirection and Request Dispatcher. We said that Response Redirection was used to redirect response of the Servlet to another application resource. This resource might be anotherServlet or any JSP page.

Two forms of Response redirection were discussed. These were:

  1. Sending a standard request:
     Using response.sendRedirect("path of resource") method, a new request is generated which redirects the user to the given URL. If the URL is of another servlet, that second servlet will not be able to access the original request object.

  2. Redirection to an error page:
     An error code is passed as a parameter along with message to response.sendRedirect(int, msg) method. This method redirects the user to the particular error page in case of occurrence of specified error.

Similarly request dispatching provides us the facility to forward the request processing to another servlet, or to include the output of another resource (servlet, JSP or HTML etc) in the response. Unlike Response Redirection, request object of calling resource is available to called resource. The two ways of Request Dispatching are:

  1. Forward:
     Forwards the responsibility of request processing to another resource.

  2. Include:
     Allows a servlet to include the results of another resource in its response. So unlike forward, the first servlet to receive the request is the one which finishes the response.

Example Code: Request Dispatching – Include

Let's start with the example of include. We will see how a Servlet includes the output of another resource in its response. The following example includes a calling Servlet MyServlet and Servlet IncludeServlet, who’s output will be included in the calling Servlet.

The code of MyServlet.java servlet is given below.
MyServlet.java

import java.io.*;
import java.net.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class MyServlet extends HttpServlet {

    /* this method is being called by both doGet() and doPost(). We
     * usually follow this practice, when we are not sure about the
     * type of incoming request to the servlet. So the actual
     * processing is being done in the processRequest().
     */
    protected void processRequest(HttpServletRequest request, ServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<body>");
        out.println("<h1>Start of include request </h1>");
        out.flush();

        // getting the object of ServletContext, that will be used to
        // obtain the object of RequestDispatcher
        ServletContext context = getServletContext();

        // getting the object of RequestDispatcher by passing the path
        // of included resource as a parameter
        RequestDispatcher rd = context.getRequestDispatcher("/includeservlet");
// calling include method of RequestDispatcher by passing
// request and response objects as parameters. This will execute
// the second servlet and include its output in the first servlet
rd.include(request, response);

/* the statements below will be executed after including the
output of the /includeservlet */
out.println("<h1>End of
out.println("</body>");
out.println("</html>");

// closing PrintWriter stream
out.close();
}

// This method only calls processRequest()
protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    processRequest(request, response);
}

// This method only calls processRequest()
protected void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    processRequest(request, response);
}

} // end MyServlet
Include Servlet
Now lets take a look at the code of IncludeServlet.java

import java.io.*;
import java.net.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class IncludeServlet extends HttpServlet {
    // this method is being called by both doGet() and doPost()
    protected void processRequest(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {

        // Obtaining the object of PrintWriter, this will return the
        // same PrintWriter object we have in MyServlet
        PrintWriter out = response.getWriter();

        // Including a HTML tag using PrintWriter
        out.println("<h1> <marquee>I am included </marquee></h1>");
    }

    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        processRequest(request, response);
    }

    protected void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
    }
} // end IncludeServlet

In the processRequest(), firstly we get the PrintWriter stream from the HttpServletResponse object. Then we include an HTML tag to the output of the calling servlet. One thing that must be considered is that PrintWriter stream is not closed in the end, because it is the same stream that is being used in the calling servlet and this stream may also be used in the calling servlet again. So, if it is closed over here, it can not be used again in the calling servlet.
web.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<web-app>
    <servlet>
        <servlet-name>MyServlet</servlet-name>
        <servlet-class>MyServlet</servlet-class>
    </servlet>
    <servlet>
        <servlet-name>IncludeServlet</servlet-name>
        <servlet-class>IncludeServlet</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>MyServlet</servlet-name>
        <url-pattern>/myservlet</url-pattern>
    </servlet-mapping>
    <servlet-mapping>
        <servlet-name>IncludeServlet</servlet-name>
        <url-pattern>/includeservlet</url-pattern>
    </servlet-mapping>
</web-app>
```
Code Example: Request Dispatcher - forward

As discussed earlier, we can forward the request processing to another resource using forward method of request dispatcher. In this example, the user enters his/her name and salary on the index.html and submits the form to FirstServlet, which calculates the tax on salary and forwards the request to another servlet for further processing i.e. SecondServlet.

index.html

```html
<html>
<body>

<form method="POST" ACTION = "firstservlet" NAME="myForm">

<h2>Enter your name</h2>

<INPUT TYPE="text" name="name"/>
<br/>

<h2>Salary</h2>

<INPUT TYPE="text" name="salary"/>
<br/> <BR/> <BR/>

<INPUT type="submit" value="Submit"/>

</form>

</body>
</html>
```

FirstServlet.java

```java
import java.io.*;
import java.net.*;
import javax.servlet.*;
import javax.servlet.http.*;

class FirstServlet extends HttpServlet {

    // this method is being called by both doGet() and doPost()
    protected void processRequest(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {

        // getting value of salary text field of the HTML form
        String salary = request.getParameter("salary");
```
// converting it to the integer.
int sal = Integer.parseInt(salary);

// calculating 15% tax
int tax = (int)(sal * 0.15);

// converting tax into string
String taxValue = tax + "";

// request object can store values in key-value form, later it
// can be retrieved by using getAttribute() method
request.setAttribute("tax", taxValue);

// getting object of servletContext
ServletContext sContext = getServletContext();

// getting object of request dispatcher
RequestDispatcher rd = sContext.getRequestDispatcher("/secondservlet");

// calling forward method of request dispatcher
rd.forward(request, response);

// This method is calling processRequest()
protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    processRequest(request, response);
}

// This method is calling processRequest()
protected void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    processRequest(request, response);
}

Note: In the case of Forward, it is illegal to make the reference of PrintWriter stream in the calling Servlet. Only the called resource can use PrintWriter stream to generate response.
SecondServlet.java

```java
import java.io.*;
import java.net.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class SecondServlet extends HttpServlet {
    // this method is being called by both doGet() and doPost()
    protected void processRequest(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        // obtaining values of name and salary text fields of index.html
        String name = request.getParameter("name");
        String salary = request.getParameter("salary");

        /* getting attribute value that has been set by the calling
       servlet i.e. FirstServlet */
        String tax = (String)request.getAttribute("tax");

        // generating HTML tags using PrintWriter
        out.println("<html>");
        out.println("<head>");
        out.println("<title>SecondServlet</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("<h1> Welcome " + name + "</h1>");
        out.println("<h3> Salary " + salary + "</h3>");
```

© Copyright Virtual University of Pakistan
out.println("<h3> Tax " + tax + "</h3>");
out.println("</body>");
out.println("</html>");
out.close();

} // This method is calling processRequest()
protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    processRequest(request, response);
}

} // This method is calling processRequest()
protected void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    processRequest(request, response);
}

}

web.xml

<?xml version="1.0" encoding="UTF-8"?>
<web-app>
    <servlet>
        <servlet-name>FirstServlet</servlet-name>
        <servlet-class>FirstServlet</servlet-class>
    </servlet>

    <servlet>
        <servlet-name>SecondServlet</servlet-name>
        <servlet-class>SecondServlet</servlet-class>
    </servlet>
<servlet-mapping>
  <servlet-name>FirstServlet</servlet-name>
  <url-pattern>/firstservlet</url-pattern>
</servlet-mapping>

<servlet-mapping>
  <servlet-name>SecondServlet</servlet-name>
  <url-pattern>/secondservlet</url-pattern>
</servlet-mapping>
</web-app>

HttpServletRequest Methods

Let’s discuss some methods of HttpServletRequest class

setAttribute(String, Object)

We can put any object to the context using setAttribute() method in the key-value pair form. These attributes are also set or reset between requests. These are often used in conjunction with Request Dispatcher. This has also been illustrated in the above example. These attributes are available everywhere in the same web application so that any other Servlet or JSP resource can access them by using getAttribute() method.

getAttribute(String)

The objects set by the setAttribute() method can be accessed using getAttribute() method. Passing the key in the form of string as a parameter to this method will return the object associated with that particular key in the context. Cast the object into its appropriate type.

getMethod()

This method returns the name of HTTP method which was used to send the request. The two possible returning values could be, get or post.

getRequestURL()

It can be used to track the source of Request. It returns the part of the request’s URL with out query string.
getProtocol()

It returns the name and version of the protocol used.

getHeaderNames()

It returns the enumeration of all available header names that are contained in the request.

getHeaderName()

It takes a String parameter that represents the header name and returns that appropriate header. Null value is returned if there is no header exists with the specified name.

HttpServletRequest Methods

Let’s discuss some methods of HttpServletRequest class

setContentType()

Almost every Servlet uses this header. It is used before getting the PrintWriter Stream. It is used to set the Content Type that the PrintWriter is going to use. Usually we set “text/html”, when we want to send text output or generate HTML tags on the client’s browser.

setContentLength()

This method is used to set the content length. It takes length as an integer parameter.

addCookie()

This method is used to add a value to the Set-Cookie header. It takes a Cookie object as a parameter and adds it to the Cookie-header. We will talk more about Cookies in the session tracking part.

sendRedirect()

This method redirects the user to the specific URL. This method also accepts the relative URL. It takes URL string as parameter and redirects the user to that resource.
Session Tracking

Many applications require a series of requests from the same client to be associated with one another. For example, any online shopping application saves the state of a user's shopping cart across multiple requests. Web-based applications are responsible for maintaining such state, because HTTP protocol is stateless. To support applications that need to maintain state, Java Servlet technology provides an API for managing sessions and allows several mechanisms for implementing sessions.

Before looking inside the session tracking mechanism lets see the limitation of HTTP protocol to get the real picture of problems that can happen with out maintaining the session.

**Continuity problem- user's point of view**

Suppose a user logs on to the online bookshop, selects some books and adds them to his cart. He enters his billing address and finally submits the order. HTTP cannot track session as it is stateless in nature and user thinks that the choices made on page1 are remembered on page3.
The server has a very different point of view. It considers each request independent from other even if the requests are made by the same client.

References

- Java A Lab Course by Umair Javed
- Core Servlet and JSP by Marty Hall
Session Tracking
We have discussed the importance of session tracking in the previous handout. Now, we’ll discover the basic techniques used for session tracking. Cookies are one of these techniques and remain our focus in this handout. Cookies can be used to put small information on the client’s machine and can be used for various other purposes besides session tracking. An example of simple “Online Book Store”, using cookies, will also be surveyed.
As mentioned elsewhere, HTTP is a stateless protocol. Every request is considered independent of every other request. But many applications need to maintain a conversational state with the client. A shopping cart is a classical example of such conversational state.

Store State Somewhere

To maintain the conversational state, the straightforward approach is to store the state. But where? These states either can be stored on server or on client. However, both options have their merits and demerits. Let’s cast a glance on these options:
Storing state on server side makes server really complicated as states needed to be stored for each client. Some one can imagine how much space and processing is required in this scenario as some web servers are hit more than hundred times in a second. E.g. Google, Yahoo etc.
What if states are stored on client side in order to maintain a conversation? Do all the clients permit you doing that? What if client (user) wiped out these states from the machine?
Concluding this discussion, state is stored neither completely on server side nor on client. States are maintained by the mutual cooperation of both client & server. Generally modern servers give the capability to store state on the server side and some information (e.g. client ID/state ID) passed from the client will relate each client with its corresponding state.

Post–Notes

In order to maintain the conversational state, server puts little notes (some text, values etc) on the client slide. When client submits the next form, it also *unknowingly* submits these little notes. Server reads these notes and able to recall who the client is.

Three Typical Solutions

Three typical solutions come across to accomplish session tracking. These are:
1. Cookies
2. URL Rewriting
3. Hidden Fields
Cookies

What a cookie is?
Don’t be tempted? These are not, what you might be thinking off. In fact, in computer terminology, “a cookie is a piece of text that a web server can store on a client’s (user) hard disk”.
Cookies allow the web sites to store information on a client machine and later retrieve it. The pieces of information are stored as name-value pair on the client. Later while reconnecting to the same site (or same domain depending upon the cookie settings), client returns the same name-value pair to the server.

Cookie’s Voyage
To reveal the mechanism of cookies, let’s take an example. We are assuming here that the web application we are using will set some cookies

• If you type URL of a Web site into your browser, your browser sends a request for that web page
  o For example, when you type www.amazon.com a request is send to the Amazon’s server
• Before sending a request, browser looks for cookie files that amazon has set
  o If browser finds one or more cookie files related to amazon, it will send it along with the request
  o If not, no cookie data will be sent with the request
• Amazon web server receives the request and examines the request for cookies
  o If cookies are received, amazon can use them
  o If no cookie is received, amazon knows that you have not visited before or the cookies that were previously set got expired.
  o Server creates a new cookie and send to your browser in the header of HTTP Response so that it can be saved on the client machine.

Potential Uses of Cookies

Whether cookies have more pros or cons is arguable. However, cookies are helpful in the following situations

• Identifying a user during an e-commerce session. For example, this book is added into shopping cart by this client.
• Avoiding username and password as cookies are saved on your machine
• Customizing a site. For example, you might like email-inbox in a different look form others. This sort of information can be stored in the form of cookies on your machine and latter can be used to format inbox according to your choice.
• Focused Advertising. For example, a web site can store information in the form of cookies about the kinds of books, you mostly hunt for.
Sending Cookies to Browser
Following are some basic steps to send a cookie to a browser (client).

1. Create a Cookie Object

A cookie object can be created by calling the Cookie constructor, which takes two strings: the cookie name and the cookie value.

```
Cookie c = new Cookie(“name”, “value”);
```

2. Setting Cookie Attributes
Before adding the cookie to outgoing headers (response), various characteristics of the cookie can be set. For example, whether a cookie persists on the disk or not. If yes then how long.
A cookies by default, lasts only for the current user session (i.e. until the user quits the session) and will not be stored on the disk.
Using setMaxAge(int lifetime) method indicates how much time (in seconds) should elapse before the cookie expires.

```
c.setMaxAge(60); // expired after one hour
```

3. Place the Cookie into HTTP response
After making changes to cookie attributes, the most important and unforgettable step is to add this currently created cookie into response. If you forget this step, no cookie will be sent to the browser.

```
response.addCookie(c);
```

Reading Cookies from the Client
To read the cookies that come back from the client, following steps are generally followed.

1. Reading incoming cookies

To read incoming cookies, get them from the request object of the HttpServletRequest by calling following method

```
Cookie cookies[] = request.getCookies();
```

This call returns an array of Cookies object corresponding to the name & values that came in the HTTP request header.

2. Looping down Cookies Array
Once you have an array of cookies, you can iterate over it. Two important methods of Cookie class are

```
getName() & getValue(). These are used to retrieve cookie name and value respectively.
```
Example Code 1: Repeat Visitor

In the example below, servlet checks for a unique cookie, named “repeat”. If the cookie is present, servlet displays “Welcome Back”. Absence of cookie indicates that the user is visiting this site for the first time thus servlet displays a message “Welcome Aboard”.

This example contains only one servlet “RepeatVisitorServlet.java” and its code is given below. A code snippet of web.xml is also accompanied.

Note: As a reminder, all these examples are built using netBeans 4.1. This IDE will write web.xml for you. However, here it is given for your reference purpose only, or for those which are not using any IDE to strengthen their concepts

RepeatVisitorServlet.java

```
import java.io.*;import java.net.*;import javax.servlet.*;import javax.servlet.http.*;
public class RepeatVisitorServlet extends HttpServlet {
    // Handles the HTTP <code>GET</code> method.
    protected void doGet(HttpServletRequest request,
                          HttpServletResponse response) throws ServletException, IOException {
        processRequest(request, response);
    }

    // Handles the HTTP <code>POST</code> method.
    protected void doPost(HttpServletRequest request,
                          HttpServletResponse response) throws ServletException, IOException {
        processRequest(request, response);
    }

    // called from both doGet() & doPost()
    protected void processRequest(HttpServletRequest request,
```

© Copyright Virtual University of Pakistan
Web Design & Development – CS506

HttpServletResponse response) throws ServletException, IOException {

    response.setContentType("text/html");

    PrintWriter out = response.getWriter();
    // writing html
    out.println("<html>");
    out.println("<body>");

    out.println("<h2>Cookie Example </h2>");
    String msg = "";
    boolean repeatVisitor = false;

    // reading cookies
    Cookie[] cookies = request.getCookies();

    // if cookies are returned from request object
    if (cookies != null) {
        // search for cookie -- repeat
        for (int i = 0; i < cookies.length; i++) {
            // retrieving one cookie out of array
            Cookie c = cookies[i];

            // retrieving name & value of the cookie
            String name = c.getName();
            String val = c.getValue();

            // confirming if cookie name equals “repeat” and
            // value equals “yes”
            if (name.equalsIgnoreCase("repeat") && val.equalsIgnoreCase("yes")) {

                msg = "Welcome Back"); repeatVisitor = true; break;
            }
        }
    } // end for
    } // end if

    // if no cookie with name “repeat” is found if (repeatVisitor == false) {
        // create a new cookie
        Cookie c1 = new Cookie("repeat", "yes");

        // setting time after which cookies expires
        c1.setMaxAge(60);

        // adding cookie to the response
        response.addCookie(c1);

        msg = "Welcome Aboard";
    }
}
// displaying message value
out.println("<h2>" + msg + "</h2>);

out.println("</body>");
out.println("</html>");
out.close();} // end RepeatVisitorServlet

web.xml
<?xml version="1.0" encoding="UTF-8"?>
<web-app>
    <servlet>
        <servlet-name>RepeatVisitorServlet</servlet-name>
        <servlet-class>RepeatVisitorServlet</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>RepeatVisitorServlet</servlet-name>
        <url-pattern>/repeatexamp</url-pattern>
    </servlet-mapping>
</web-app>

Output
On first time visiting this URL, an output similar to the one given below would be displayed

Welcome Aboard

On refreshing this page or revisiting it within an hour (since the age of cookie was set to 60 mins), following output should be expected.

Welcome Back
Example Code2: Online Book Store using cookies
A scale down version of online book store is going to be built using cookies. For the first time, cookies will be used to maintain the session.
Three books will be displayed to the user along with check boxes. User can select any check box to add the book in the shopping cart. The heart of the application is, it remembers the books previously selected by the user.
The following figure will help you understand the theme of this example. Books displayed under the heading of “You have selected the following books” were added to cart one after another. The important thing is server that remembers the previously added books by the same user and thus maintains the session. Session management is accomplished using cookies.

Online Book Store

- [ ] java core servlets
- [ ] java how to program
- [ ] java complete reference

Add to Cart

You have selected following books

java core servlets
java complete reference

Online Book Store example revolves around one ShoppingCartServlet.java. This Servlet has one global HashMap (globalMap) in which HashMap of individual user (sessionInfo) are going to be stored. This (sessionInfo) HashMap stores the books selected by the user.
What’s the part of cookies? Cookie (named JSESSIONID, with unique value) is used to keep the unique sessionID associated with each user. This sessionID is passed back and forth between user and the server and is used to retrieve the HashMap (sessionInfo) of the user from the global HashMap at the server. It should be noted here that, HashMaps of individual users are stored in a global HashMap against a sessionID.
ShoppingCartServlet.java
import java.io.*;import java.net.*;import javax.servlet.*;import javax.servlet.http.*;
public class ShoppingCartServlet extends HttpServlet {

    // used to generate a unique value which is
    // used as a cookie value

    public static int S_ID = 1;

    // used to store HashMaps of individual users
    public static HashMap globalMap = new HashMap();

    // Handles the HTTP GET method.
    protected void doGet(HttpServletRequest request,
                HttpServletResponse response)throws ServletException, IOException {
            processRequest(request, response);
    }

    // Handles the HTTP <code>POST</code> method.
    protected void doPost(HttpServletRequest request,
                HttpServletResponse response)throws ServletException, IOException {
            processRequest(request, response);
    }

    // called from both doGet() & doPost()
    protected void processRequest(HttpServletRequest request,
                HttpServletResponse response)
    throws ServletException, IOException {

        response.setContentType("text/html;charset=UTF-8");
        // declaring user's HashMap
        HashMap sessionInfo = null;String sID = "";

        // method findCookie is used to determine whether browser// has send any cookie named
        "JSESSIONID"
        Cookie c = findCookie(request);

        // if no cookies named "JSESSIONID" is received, means that// user is visiting the site for the first time.
        if (c == null) {
            // make a unique string
            sID = makeUniqueString();
        }
    }
// creating a HashMap where books selected by the user will be stored
sessionInfo = new HashMap();
// add the user's HashMap (sessionInfo) into the globalMap against unique string i.e. sID
globalMap.put(sID, sessionInfo);

// create a cookie named "JSESSIONID" alongwith value of sID i.e. unique string
Cookie sessionCookie = new Cookie("JSESSIONID", sID);

// add the cookie to the response
response.addCookie(sessionCookie);
}
else {
// if cookie is found named "JSESSIONID", retrieve a HashMap from the globalMap against cookie value i.e. unique string which is your sessionID

sessionInfo = (HashMap) globalMap.get(c.getValue());
}

PrintWriter out = response.getWriter();

out.println("<html> ");
out.println("<head> ");
out.println("<title>Shopping Cart Example</title> ");
out.println("</head> ");
out.println("<body> ");
out.println("<h1>Online Book Store</h1> ");

String url = "http://localhost:8084/cookiesessionex/shoppingcartex";

// user will submit the from to the same servlet

out.println("<form action=" + url + ">
" + 
"<input type=checkbox name=firstCB value=firstCB />
" + " java core servlets</h3>
" + 
"<br>
" + 
"<h3>
" + " java how to program</h3>
" + 
"<br>
" + 
"<h3>
" + " java complete reference</h3>
" + 
"<br>
" + 
"<input type=submit value="Add to Cart" />
" + 
"</form>
");

out.println("<br/>");
out.println("<h1>You have selected followig books</h1>");
out.println("<br/>");

© Copyright Virtual University of Pakistan 305
public class Cookie {  
  String name;  
  String value;  
  public void setName(String name) { this.name = name; }  
  public void setValue(String value) { this.value = value; }  
}

String fBook = request.getParameter("firstCB");  
String sBook = request.getParameter("secondCB");  
String tBook = request.getParameter("thirdCB");

public String makeUniqueString() { return "ABC" + S_ID++;
  }

public static HashMap<String, String> findTableStoringSessions() { return globalMap;
  }

public Cookie findCookie(HttpServletRequest request) {  
  Cookie[] cookies = request.getCookies();  
  if (cookies != null) {  
    for(int i=0; i<cookies.length; i++) {  
      Cookie c = cookies[i];  
      if (c.getName() .equals("JSESSIONID")) { // doSomethingWithcookiereturn c;  
    }  
  }  
  return null;

printSessionInfo(out, sessionInfo);
out.println("</body>");  
out.println("</html>");  
out.close();
} // end processRequest() // method used to generate a unique string

// retrieveing params of check boxes

if (fBook != null & & fBook.equals("firstCB") ) {  
  sessionInfo.put("firstCB", "java core servlets");
}  

if (sBook != null & & sBook.equals("secondCB") ) {sessionInfo.put("secondCB",  
  "java how to program");
}  

if (tBook != null & & tBook.equals("thirdCB") ) {sessionInfo.put("thirdCB", "java  
  complete reference");
}  

// used to display the books currently stored in  
// the user's HashMap i.e. sessionInfo

// to retrieve the session info from the request

// used to print the books currently stored in // user's HashMap i.e. sessionInfo
public void printSessionInfo(PrintWriter out, HashMap sessionInfo) {
    String title = "";
    title = (String) sessionInfo.get("firstCB");
    if (title != null) {
        out.println("<h3>" + title + "</h3>");
    }
    title = (String) sessionInfo.get("secondCB");
    if (title != null) {
        out.println("<h3>" + title + "</h3>");
    }
    title = (String) sessionInfo.get("thirdCB");
    if (title != null) {
        out.println("<h3>" + title + "</h3>");
    }
} // end ShoppingCartServlet

web.xml

<?xml version="1.0" encoding="UTF-8"?>
<web-app>
    <servlet> <servlet-name> ShoppingCart </servlet-name> <servlet-class> ShoppingCartServlet </servlet-class> </servlet>
    <servlet-mapping>
        <servlet-name> ShoppingCart </servlet-name> <url-pattern> /shoppingcartex </url-pattern>
    </servlet-mapping>
</web-app>

References:
- Java A Lab Course by Umair Javed
- Core Servlets and JSP by Marty Hall
- Stanford Course – Internet Technologies
- Java API documentation
Session Tracking 2
In the last handout we have discussed the solutions for session tracking and talked about one important mechanism cookies in detail. We said cookies allow the server to store information on a client machine and later retrieve it. Now we will see two more mechanisms that provide us facility to maintain a session between user’s requests. These are URL Rewriting and Hidden Form Fields. After that we will discuss a session tracking API provided by java.

URL Rewriting
URL rewriting provides another way for session tracking. With URL rewriting, the parameter that we want to pass back and forth between the server and client is appended to the URL. This appended information can be retrieve by parsing the URL. This information can be in the form of:
- Extra path information,
- Added parameters, or
- Some custom, server-specific URL change

Note: Due to limited space available in rewriting a URL, the extra information is usually limited to a unique session ID.
The following URLs have been rewritten to pass the session ID 123
- Original –  http://server: port/servlet /rewrite
- Extra path information – http://server:port/servlet/rewrite/123
- Added parameters – http://server:port/servlet/rewrite?id=123
- Custom change – http://server:port/servlet/rewrite;sid$123

Disadvantages of URL rewriting
The following Disadvantages of URL rewriting, are considerable: -
- What if the user bookmarks the page and the problem get worse if server is not assigning a unique session id.
- Every URL on a page, which needs the session information, must be rewritten each time page is served, which can cause
  o Computationally expensive
  o Can increase communication overhead
- Unlike cookies, state information stored in the URL is not persistent
- This mechanism limits the client interaction with the server to HTTP GET request.

Example Code: OnlineBookStore using URL Rewriting
This is the modified version of online book store (selling two books only, however you can add in on your own) that is built using cookies in the last handout. Another important difference is books are displayed in the form of hyperlink instead of check boxes. URL rewriting mechanism is used to maintain session information.

How to make Query String
Before jumping on to example, one important technique is needed to be learned i.e. making on query string. If you ever noticed the URL of a servlet in a browser that is receiving some HTML form values, also contains the HTML fields name with values entered/selected by the user.
Now, if you want to pass some attribute and values along with URL, you can use the technique of query string.
Attribute names and values are written in pair form after the ?. For example, if you want to send attribute “name” and its value “ali”, the URL will look like

- Original URL
  
  http://server:port/servletex/register
- After adding parameters
  
  http://server:port/servletex/register?name=ali

If you want to add more than one parameter, all subsequent parameters are separated by & sign. For example

- Adding two parameters –

  http://server:port/servletex/register?name=ali&address=gulberg

**URLRewriteServlet.java**

```java
import java.io.*;import java.net.*;import javax.servlet.*;import javax.servlet.http.*;
public class URLRewriteServlet extends HttpServlet {

  // used to generate a unique value which is also used as a cookie value

  public static int S_ID = 1;

  public static HashMap globalMap = new HashMap();

  // Handles the HTTP GET method.
  protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
    processRequest(request, response);
  }

  // Handles the HTTP <code>POST</code> method.
  protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
    processRequest(request, response);
  }

  // called from both doGet() & doPost()protected void
  processRequest(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
    // declaring user's HashMap
    HashMap sessionInfo = null;
    // reading sessionID
    String sID = request.getParameter("JSESSIONID");
    /* if parameter JSESSIONID is received, means that user is visiting the site for the first time. */
    if (sID == null) {
      // make a unique string
    }
```
sID = makeUniqueString();
// creating a HashMap where books selected by the
// user will be stored

sessionInfo = new HashMap();
// add the user's HashMap (sessionInfo) into the
// globalMap against unique string i.e. sID

globalMap.put(sID, sessionInfo);
} else {
// if parameter "JSESSIONID" has some value
// retrieve a HashMap from the globalMap against
// sID i.e. unique string which is your sessionID

sessionInfo = (HashMap)globalMap.get(sID);
}

response.setContentType("text/html;charset=UTF-8");
PrintWriter out = response.getWriter();

out.println("<html>");
out.println("<head>");
out.println("<title>Shopping Cart Example</title>");
out.println("<head>");
out.println("<body>");
out.println("<h1>Online Book Store</h1>");

// Making three URLs by using query string mechanism
// The attributes/parameters are JSSESSIONID and
// book name (like firstCB) along with values sID and book name respectively
+ sID + "&firstCB=firstCB";
+ sID + "&secondCB=secondCB";
out.println("<h3><a href=" + firsturl + ">" + " java core servlts </a> </h3>
+"<br/>
"+<h3><a href=" + secondurl + ">" +
" java how to program </a> </h3>
+"<br/>

);
out.println("<br/>");
out.println("<h1>You have selected following books</h1>");
out.println("<br/>");

//retrieving params that are embebed in URLs

String fBook = request.getParameter("firstCB");String sBook = request.getParameter("secondCB");
// if first book is selected then add it to
// user's HashMap i.e. sessionInfo

if (fBook != null && fBook.equals("firstCB")) {
    sessionInfo.put("firstCB", "java core servlets");
}

// if second book is selected then add it to
// user's HashMap i.e. sessionInfo

if (sBook != null && sBook.equals("secondCB")) {
    sessionInfo.put("secondCB", "java how to program");
}

// used to display the books currently stored in
// the user's HashMap i.e. sessionInfo

printSessionInfo(out, sessionInfo);
out.println("</body>");
out.println("</html>");
out.close();
} // end processRequest()

// method used to generate a unique string
public String makeUniqueString() { return "ABC" + S_ID++;
}

// returns a reference global HashMap.
//public static HashMap findTableStoringSessions() { return globalMap; }

// used to print the books currently stored in// user's HashMap i.e. sessionInfo
public void printSessionInfo(PrintWriter out, HashMap sessionInfo) {
    String title = "";
    title = (String)sessionInfo.get("firstCB");
    if (title != null) {
        out.println("<h3>" + title +"</h3>";)
        title = (String)sessionInfo.get("secondCB");
    if (title != null) {
        out.println("<h3>" + title +"</h3>";
    }
}
} // end URLRewriteServlet
web.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<web-app>
    <servlet> <servlet-name>URLRewriteServlet</servlet-name> <servlet-class>URLRewriteServlet</servlet-class>
</servlet>
    <servlet-mapping> <servlet-name>URLRewriteServlet</servlet-name> <url-pattern>/*</url-pattern>
</servlet-mapping>
</web-app>
```

Hidden Form Fields

HTML forms can have an element that looks like the following:

```html
<INPUT TYPE="HIDDEN" NAME="sessionid" VALUE="123" />
```

Hidden Forms Fields do not affect the appearance of HTML page. They actually contain the information that is needed to send to the server. Thus, hidden fields can also be used to store information (like sessionid) in order to maintain session.

In the above figure you can see the use of Hidden form fields for storing particular information.

Java Solution for Session Tracking

Java provides an excellent solution to all the problems that occurred in tracking a session. The Servlet API provides several methods and classes specifically designed to handle session tracking. In other words, servlets have built in session tracking.

Sessions are represented by an HttpSession object. HttpSession tacking API built on top of URL rewriting and cookies. All cookies and URL rewriting mechanism is hidden and most application server uses cookies but automatically revert to URL rewriting when cookies are unsupported or explicitly disabled. Using HttpSessionAPI in servlets is straightforward and involves looking up the session object associated with the current request, creating new session object when necessary, looking up information associated with a session,
storing information in a session, and discarding completed or abandoned sessions.

**Working with HttpSession**

Let’s have a look on HttpSession working step by step.

1. **Getting the user’s session object**

   To get the user’s session object, we call the `getSession()` method of `HttpServletRequest` that returns the object of `HttpSession`

   ```java
   HttpSession sess = request.getSession(true);
   ```

   If `true` is passed to the `getSession()` method, this method returns the current session associated with this request, or, if the request does not have a session, it creates a new one. We can confirm whether this session object (`sess`) is newly created or returned by using `isNew()` method of `HttpSession`. In case of passing `false`, null is returned if the session doesn’t exist.

2. **Storing information in a Session**

   To store information in Session object (`sess`), we use `setAttribute()` method of `HttpSession` class. Session object works like a HashMap, so it is able to store any java object against key. So you can store number of keys and their values in pair form. For example,

   ```java
   sess.setAttribute(“sessionid”, ”123”);
   ```

3. **Looking up information associated with a Session**

   To retrieve back the stored information from session object, `getAttribute()` method of `HttpSession` class is used. For example,

   ```java
   String sid=(String)sess.getAttribute(“sessionid”);
   ```

   **Note:** `getAttribute()` method returns `Object` type, so `typecast` is required.

4. **Terminating a Session**

   After the amount of time, session gets terminated automatically. We can see its maximum activation time by using `getMaxInactiveInterval()` method of `HttpSession` class. However, we can also terminate any existing session manually. For this, we need to call `invalidate()` method of `HttpSession` class as shown below.

   ```java
   sess.invalidate()
   ```

**Example Code: Showing Session Information**

To understand HttpSession API properly we need to have a look on an example. In this example, we will get the session object and check whether it is a new user or not. If the user is visiting for the first time, we will print “Welcome” and if we find the old one, we’ll print “Welcome Back”. Moreover, we will print the session information and count the number of accesses for every user

```java
import java.io.*;
import java.net.*;
```
import javax.servlet.*;
import javax.servlet.http.*;

public class ShowSessionServlet extends HttpServlet {

    // Handles the HTTP <code>GET</code> method.
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException
    {
        processRequest(request, response);
    }

    // Handles the HTTP <code>POST</code> method.
    protected void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException
    {
        processRequest(request, response);
    }

    // called from both doGet() & doPost()
    protected void processRequest(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException
    {
        // used for displaying message (like Welcome, Newcomer) to user
        private String heading;
        response.setContentType("text/html");

        // Getting session object
        HttpSession session = request.getSession(true);

        /* Getting stored information using getAttribute() method */
Integer accessCount = (Integer)session.getAttribute("sessionCount");

/* If user comes for the first time, accessCount will be
assigned null, so we can guess easily that this a new user */
if (accessCount == null)
{
    accessCount = new Integer(1);
    heading = "Welcome, Newcomer";
}
else
{
    heading = "Welcome Back";

    // Incrementing the value
    accessCount = new Integer(accessCount.intValue() + 1);
}

/* Storing the new value of accessCount in the session using
setAttribute() method */
session.setAttribute("sessionCount", accessCount);

// Getting the PrintWriter
PrintWriter out = response.getWriter();

/*Generating HTML tags using PrintWriter to print session info
and no of times this user has accessed this page */
out.println("<HTML>" + " <BODY>" + " <h1>Session Tracking Example</h1>" +
    " <h2>Information on Your Session:</h2>" +
    " <H3> Session ID: " + session.getId() + "</H3>" +
    " </BODY> </HTML>");
"<H3>Number of Previous Accesses: " + accessCount +
"</H3>" +
"</BODY>" +
"</HTML>");

//Closing the PrintWriter stream
out.close();
}
} // end processRequest
} // end ShowSessionServlet class

web.xml

<web-app>
   <servlet> <servlet-name> ShowSession </servlet-name> <servlet-class> ShowSessionServlet </servlet-class>
   </servlet>

   <servlet-mapping>
      <servlet-name> ShowSession </servlet-name>
      <url-pattern> /showsession </url-pattern>
   </servlet-mapping>
</web-app>

HttpSession – Behind the scenes

When we call getSession() method, there is a lot going on behind the scenes. For every user, a unique session ID is assigned automatically. As the server deals with lot of users at a time, this ID is used to distinguish one user from another. Now here is the question, how this ID sends to the user? Answer is, there are two options

- **Option 1**: If the browser supports cookies, the Servlet will automatically creates a session cookie and store the session ID within that cookie.
- **Option 2**: If the first option fails because of browser that does not support cookies then the Servlet will try to extract the session ID from the URL.
Encoding URLs sent to Client

Servlet will automatically switch to URL rewriting when cookies are not supported or disabled by the client. When Session Tracking is based on URL rewriting, it requires additional help from the Servlets. For a Servlet to support session tracking via URL rewriting, it has to rewrite (encode) every local URL before sending it to the client. Now see how this encoding works.

HttpServletRequest provides two methods to perform encoding:

- String encodeURL(String URL)
- String encodeRedirectURL(String URL)

If Cookies are disabled, both methods encode (rewrite) the specific URL to include the session ID and returns the new URL. However, if cookies are enabled, the URL is returned unchanged.

**Difference between encodeURL() and encodeRedirectURL()**

encodeURL() is used for URLs that are embedded in the webpage, that the servlet generates. For example,

```java
String URL = "/servlet/sessiontracker";
String eURL = response.encodeURL(URL);
out.println("<A HREF=\"" + eURL + \\"\"> ...... </A>");
```

Whereas encodeRedirectURL() is used for URLs that refers yours site is in sendRedirect() call. For example,

```java
String URL = "/servlet/sessiontracker";
String eURL = response.encodeRedirectURL(URL);
Response.sendRedirect(eURL);
```

**Example Code: OnlineBookStore using HttpSession**

This book store is a modified version of last one, which is built using URL rewriting mechanism. Here, HttpSession will be used to maintain session.

**ShoppingCartServlet.java**

```java
import java.io.*; import java.net.*; import javax.servlet.*; import javax.servlet.http.*;

public class ShoppingCartServlet extends HttpServlet {
    // Handles the HTTP GET method.
    protected void doGet(HttpServletRequest request,
                          HttpServletResponse response) throws ServletException, IOException {
        processRequest(request, response);
    }

    // Handles the HTTP <code>POST</code> method.
    protected void doPost(HttpServletRequest request,
                           HttpServletResponse response) throws ServletException, IOException {
```

© Copyright Virtual University of Pakistan

317
processRequest(request, response);
}

// called from both doGet() & doPost() protected void
processRequest(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
    response.setContentType("text/html;charset=UTF-8");

    HttpSession session = request.getSession(true);
    PrintWriter out = response.getWriter();

    out.println("<html>\n");
    out.println("<head>\n");
    out.println("<title>Shopping Cart Example</title>\n");
    out.println("<\head>\n");
    out.println("<body>\n");
    out.println("<h1>Online Book Store</h1>\n");

    // First URL built using query string, representing first book

    // Second URL built using query string, representing second book // Note that parameter name is still book,
    // so that later we need // to read only this parameter
    // Encoding URLs
    String eURL1 = response.encodeURL(firstURL); String eURL2 =
    response.encodeURL(secondURL);
    out.println("<h3><a href=" + eURL1 + ">" + " java core servlets </a> </h3>" + "<br>" +
            "<h3><a href=" + eURL2 + ">" +
            " java How to Program </a> </h3>");

    out.println("<br/>\n");
    out.println("<h1>You have selected following books</h1>\n");
    out.println("<br/>\n");

    // retrieving params that are embedded in URLs
    String fBook = request.getParameter("firstCB"); String sBook =
    request.getParameter("secondCB");
    out.println("<br/>\n");
    out.println("<h1>You have selected following books</h1>\n");
    out.println("<br/>\n");

    // retrieving param that is embedded into URL
    String book = request.getParameter("book");
    if (book != null) {
        // if firstURL, value of first hyperlink is clicked // then storing the book into session object against
fBook
if (book.equals("first")){session.setAttribute("fBook", "java core
servlets");}
// if second URL, value of second hyperlink is clicked// then storing the book into session object
against sBook
else if(book.equals("second")){session.setAttribute("sBook", "java
how to program");}
//outer if ends

// used to display the books currently stored in

// the HttpSession object i.e. session

printSessionInfo(out, session);
out.println("</body>");
out.println("</html>");
out.close();

} // end processRequest()
// used to display values stored in HttpSession object
public void printSessionInfo(PrintWriter out, HttpSession session) { String title = "";

// reading value against key fBook from session, // if exist displays it
title=(String)session.getAttribute("fBook");
if (title != null){
  out.println("<h3>" + title +"</h3>");
}

// reading value against key sBook from session, // if exist displays it
title=(String)session.getAttribute("sBook");
if (title != null){
  out.println("<h3>" + title +"</h3>");
}

} // end printSessionInfo

} // end ShoppingCartServlet

web.xml

<?xml version="1.0" encoding="UTF-8"?>
<web-app>
  <servlet> <servlet-name> ShoppingCartServlet </servlet-name>
    <servlet-class> ShoppingCartServlet </servlet-class>
  </servlet>
</web-app>
Some Methods of HttpSession

Now let’s explore some methods of HttpSession class

- **setAttribute(String, Object)**
  This method associates a value with a name.

- **getAttribute(String)**
  Extracts previously stored value from a session object. It returns null if no value is associated with the given name.

- **removeAttribute(String)**
  This method removes values associated with the name.

- **getId()**
  This method returns the unique identifier of this session.

- **getCreationTime()**
  This method returns time at which session was first created.

- **getMaxInactiveInterval()**, `setMaxInactiveInterval(int)`
  To get or set the amount of time session should go without access before being invalidated.

References:

- Java A Lab Course by Umair Javed
- Core Servlets and JSP by Marty Hall
- Stanford Course – Internet Technologies
- Java Tutorial on Servlets


- Java API documentation
AddressBook Case Study

Using Sevlets

Design Process

In this handout, we will discuss the design process of a simple address book. A step by step procedure of creating a simple address book is given below.

Layers & Web Application

As discussed previously, normally web applications are partitioned into logical layers. Each layer performs a specific functionality which should not be mixed with other layers. For example data access layer is used to interact with database and we do not make any direct calls to database from the presentation layer. Layers are isolated from each other to reduce coupling between them but they provide interfaces to communicate with each other.

Simplified view of a web application and its layers

-Presentation Layer

Provides a user interface for client to interact with application. This is the only part of application visible to client.

-Business Layer

The business or service layer implements the actual business logic or functionality of the application. For example in case of online shopping systems this layer handles transaction management.

- Data Layer

This layer consists of objects that represent real-world business objects such as an Order, OrderLineItem, Product, and so on. It also encapsulates classes which are used to interact with the data providing services such as databases, other web services etc.

In our case study of address book, we will also try to make use of the layered architecture. We will create a separate layer for working with data, and our presentation and business logic will be merged into sevlets. It means that we will not have separate layers for presentation and business rather one layer (formed by sevlets) will do the job of both presentation and business logic. The extent to which you divide your application into layers depends upon the size of the application and some other factors such as scalability, portability etc.
Step 1
- Create a database (AddressBook)
- Make a table named Person according to the figure shown below. It has columns name, address, phoneNum

![Person Table Image]

Step 2
The next step is to create a class that can hold the information of a single person. Remember we have stored the information in the database, now when we extract this information from the database as a result of some search, we will require some object to store the data for that particular person. The PersonInfo class will be used at that point to store the retrieved data and transport it to presentation layer. Also we extend this application and add the functionality of “AddingNewContacts” in the database. The PersonInfo class can be used to transport data from front end to the database.
- Make a PersonInfo class with the following consideration
- It has three three attributes: name, address, ph. No.
- It has a parameterized constructor which takes in the above mentioned parameters
- Override the toString() method

//File: PersonInfo.java

class PersonInfo {
    String name;
    String address;
    String phoneNum;

    PersonInfo(String n, String a, String pn) {
        name = n;
        address = a;
        phoneNum = pn;
    }

    public String toString()
    {
        // Return the string representation of the PersonInfo object
    }
}
return "Name: " + name + " Address: " + address + " Phone No: " + phoneNum;

} // end class PersonInfo

Note: To keep the code simple, attributes (name, address & phoneNum) are not declared as private, which is indeed not a good programming approach.

Step 3
Now we will create a class that will be used to interact with the database for the search, insert, update and delete operations. We will call it PersonDAO where DAO stands for the “data access object”. The PersonDAO along with the PersonInfo class forms the data layer of our application. As you can see that these two classes do not contain any code related to presentation or business logic (There is not much of business logic in this application anyway). So PersonDAO along with PersonInfo is used to retrieve and store data in this application. If at some stage we choose to use some other way of storing data (e.g. files) only the PersonDAO class will change and nothing else, which is a sign of better design as compared to a design in which we put every thing in a single class.
So, Make a PersonDAO class which contains:

- a searchPerson(String name) method that first establishes a connection to the database and returns PersonInfo object after searching the information of the specified person from the database.

//File: PersonDAO.java
import java.sql.*;

public class PersonDAO {

    // method searchPerson
    public PersonInfo searchPerson(String sName) {
        PersonInfo person = null;
        try {
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            String url = ";jdbc:odbc:AddressBookDSN";
            Connection con = DriverManager.getConnection(url);
            String sql = "SELECT * FROM Person WHERE name = ?";
            PreparedStatement pStmt = con.prepareStatement(sql);

            pStmt.setString(1, sName);
            ResultSet rs = pStmt.executeQuery();
            while (rs.next()) {
                String name = rs.getString("name");
                String address = rs.getString("address");
                String phoneNum = rs.getString("phoneNum");
                PersonInfo person = new PersonInfo(name, address, phoneNum);
            }
            return person;
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return null;
    }

} // end class PersonDAO

© Copyright Virtual University of Pakistan
pStmt.setString(1, sName);
ResultSet rs = pStmt.executeQuery();
if (rs.next()) {
    String name = rs.getString("name");
    String add = rs.getString("address");
    String pNum = rs.getString("phoneNum");
    person = new PersonInfo(name, add, pNum);
}
con.close();
} catch (Exception ex) {
    System.out.println(ex);
}
return person;
} // end method

Step 4
To find what user wants to search, we need to give user an interface through which he/she can enter the input. The SearchPersonServlet.java will do this job for us. It will collect the data from the user and submit that data to another class. The SearchPersonServlet forms the part of our presentation layer. As you can see that it is being used to present a form to the user and collect input.

- Write SearchPersonServlet.java
- Will take input for name to search in address book
- Submits the request to ShowPersonServlet

//File: SearchPersonServlet.java
import java.io.*;
import java.net.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class SearchPersonServlet extends HttpServlet {
    protected void processRequest(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<body>");
        Address Book <h1>
        </h1> + "<form action=showperson
        >" + // showperson is alias or
        // url pattern of
        // ShowPersonServlet
        <h2> Enter name to search </h2> + "<input
        type=text name=pName />
        " + "<input
        type=submit value=Search Person />
        " + "</form>
        " + "</body>");
        out.close();
    }

    // Handles the HTTP GET method.protected void
doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        processRequest(request, response);
    }
}

Step 5
The data submitted by the SearchPersonServlet will be submitted to another servlet i.e. ShowPersonServlet,
which will interact with the DataLayer(Business logic processing) collects the output and show it to the user.
The ShowPersonServlet forms the part of our presentation layer and business layer. As you can see that it is
being used to do processing on the incoming data and giving it to data layer (business layer) and present
data/output to the user (presentation layer)
    . Write ShowPersonServlet.java
    . Receives request from SearchPersonServlet
    . Instantiate objects of PersonInfo and PersonDAO class
    . Call searchPerson() method of PersonDAOclass
    . Show results

//File: ShowPersonServlet.java
import java.io.*;
import java.net.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class ShowPersonServlet extends HttpServlet {
    protected void processRequest(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {

        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        String name = request.getParameter("pName");

        // creating PersonDAO object, and calling searchPerson() method
        PersonDAO personDAO = new PersonDAO();
        PersonInfo person = personDAO.searchPerson(name);

        out.println("<html>");
        out.println("<body>");
        out.println("<h1>Search Results</h1>");

        if (person != null) {
            out.println("<h3>" + person.toString() + "</h3>");
        } else {
            out.println("<h3>Sorry! No records found</h3>");
        }

        out.println("</body>");
        out.println("</html>");
        out.close();
    }

    // Handles the HTTP GET method.
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        processRequest(request, response);
    }

    // Handles the HTTP POST method.
    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        processRequest(request, response);
    }
}
// end ShowPersonServlet
Sequence Diagram: Address Book (search use case)
Package

Many times when we get a chance to work on a small project, one thing we intend to do is to put all java files into one single directory (folder). It is quick, easy and harmless. However if our small project gets bigger, and the number of files is increasing, putting all these files into the same directory would be a nightmare for us. In java we can avoid this sort of problem by using Packages.

What is a package?

In simple terms, a set of Java classes organized for convenience in the same directory to avoid the name collisions. Packages are nothing more than the way we organize files into different directories according to their functionality, usability as well as category they should belong to. An obvious example of packaging is the JDK package from SUN (java.xxx.yyy) as shown below:

Figure. Basic structure of JDK package

Basically, files in one directory (or package) would have different functionality from those of another directory. For example, files in java.io package do something related to I/O, but files in java.net package give us the way to deal with the Network.

Packaging also helps us to avoid class name collision when we use the same class name as that of others. For example, if we have a class name called "ArrayList", its name would crash with the ArrayList class from JDK. However, this never happens because JDK use java.util as a package name for the ArrayList class (java.util.ArrayList). So our ArrayList class can be named as "ArrayList" or we can put it into another package like com.mycompany.ArrayList without fighting with anyone. The benefits of using package reflect the ease of maintenance, organization, and increase collaboration among developers. Understanding the concept of package will also help us manage and use files stored in jar files in more efficient ways.

How to create a package

Suppose we have a file called HelloWorld.java, and we want to put this file in a package world. First thing we have to do is to specify the keyword package with the name of the package we want to use (world in our case) on top of our source file, before the code that defines the real classes in the package, as shown in our HelloWorld class below:

// only comment can be here
package world;
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World");
    }
}

One thing you must do after creating a package for the class is to create nested subdirectories to represent package hierarchy of the class. In our case, we have the world package, which requires only one directory. So, we create a directory (folder) world and put our HelloWorld.java into it.

![Figure: HelloWorld in world package (C:\world\HelloWorld.java)](image)

**How to use package**

By using "import" keyword, all class files reside only in that package can be imported. For example,

```java
// we can use any public classes inside world package
import world.*;
// import all public classes from java.util package
import java.util.*;
// import only ArrayList class (not all classes in
// java.util package)
import java.util.ArrayList;
```

**Note:** While working with IDEs, You don’t have to create folders (packages) and to place classes at right locations. Many IDEs (like netBeans® 4.1) performs this job on your behalf.
# JavaServer Pages (JSP)

Like Servlets, JSP is also a specification. JSP technology enables Web developers and designers to rapidly develop and easily maintain, information-rich, dynamic Web pages that leverage existing business systems. As part of the Java technology family, JSP technology enables rapid development of Web-based applications that are platform independent. JSP technology separates the user interface from content generation, enabling designers to change the overall page layout without altering the underlying dynamic content.

## The Need for JSP

- With servlets, it is easy to
  - Read form data
  - Read HTTP request headers
  - Set HTTP status codes and response headers
  - Use cookies and session tracking
  - Share data among servlets
  - Remember data between requests
  - Get fun, high-paying jobs
- But, it sure is a pain to
  - Use those println() statements to generate HTML.
  - Maintain that HTML.

## The JSP Framework

- Use regular HTML for most of the pages
- Mark servlet code with special tags
- Entire JSP page gets translated into a servlet (once), and servlet is what actually gets invoked (for each request)
- The Java Server Pages technology combine with Java code and HTML tags in the same document to produce a JSP file.

![Java + HTML = JSP](image)

## Advantages of JSP over Competing Technologies

- **Versus ASP or ColdFusion**
  - JSPs offer better language for dynamic part i.e. Java
  - JSPs are portable to multiple servers and operating systems
- **Versus PHP**
  - JSPs offer better language for dynamic part
  - JSPs offer better tool support
- **Versus pure servlets**
  - JSPs provide more convenient way to create HTML
  - JSPs can use standard front end tools (e.g., UltraDev)
  - JSPs divide and conquer the problem of presentation and business logic.
Setting Up Your Environment

In order to create a web-application that entirely consists of JSP pages and Html based pages, the setup is fairly simple as compared to a servlet based webapplication.

- Set your CLASSPATH. No.
- Compile your code. No.
- Use packages to avoid name conflicts. No.
- Put JSP page in special directory, like WEB-INF for servlets No.
  o  tomcat_install_dir/webapps/ROOT
  o  jrun_install_dir/servers/default/default-app
- Use special URL to invoke JSP page. No
- **However**
  o  If you want to use java based classes in an application along with JSPs, previous rules about CLASSPATH, install dirs, etc, still apply to regular classes used by JSP

References:

- Java A Lab Course by Umair Javed
- Java Package Tutorial by Patrick Bouklee
  
  http://jarticles.com/package/package_eng.html
- JavaServer Pages Overview
  
Java Server Pages

As we concluded in our discussion on JSP, JSP is a text based document capable of returning either static or dynamic content to a client’s browser. Static content and dynamic content can be intermixed. The examples of static content are HTML, XML & Text etc. Java code, displaying properties of JavaBeans and invoking business logic defined in custom tags are all examples of dynamic content.

First run of a JSP

Figure below shows what phases a JSP passed through before displaying result.

The web browser makes a request to JSP source code. This code is bifurcated into HTML and java code by the JSP parser. The java source code is compiled by the Java compiler resulting in producing a servlet equivalent code of a JSP. The servlet code is intermixed with HTML and displayed to the user. It is important to note that a JSP only passes through all these phases when it is invoked for the first time or when the changes have been made to JSP. Any later call to JSP does not undergo of compilation phase.

Benefits of JSP

Convenient

- We already know java and HTML. So nothing new to be learned to work with JSP.
- Like servlets (as seen, ultimately a JSP gets converted into a servlet), provides an extensive infrastructure for
  - Tracking sessions
  - Reading and sending HTML headers
  - Parsing and decoding HTML form data

Every request for a JSP is handled by a simple JSP java thread as JSP gets converted into a servlet. Hence, the time to execute a JSP document is not dominated by starting a process.

Portable

Like Servlets, JSP is also a specification and follows a well standardized API. The JVM which is used to execute a JSP file is supported on many architectures and operating systems.
Inexpensive
There are number of free or inexpensive Web Servers that are good for commercial quality websites

JSP vs. Servlet
Let’s compare JSP and Servlet technology by taking an example that simply plays current date.
First have a look on JSP that is displaying a current date. This page more looks like a HTML page except of
two strangely written lines of codes. Also there are no signs of doGet(), doPost().

```jsp
<%@ page import="java.util.*" %>
<html>
<body>
  <h3>
    Current Date is:<%= new Date()%>
  </h3>
</body>
</html>
```

Now, compare the JSP code above with the Servlet code given below that is also displaying the current date.

```java
//File:SearchPersonServlet.java
import java.io.*;
import java.net.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;

public class SearchPersonServlet extends HttpServlet {

  protected void processRequest(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
      response.setContentType("text/html");
      PrintWriter out = response.getWriter();
      out.println("<html>
                    "<body>
                    "<h3>
                    "Current Date is:" + new Date() +
                    "</h3>
                    "</body>
                    "</html>";
    }
}
```

© Copyright Virtual University of Pakistan 333
// Handles the HTTP GET method.

protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
    processRequest(request, response);
}

// Handles the HTTP POST method.

protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
    processRequest(request, response);
}

Clearly, a lot of code is needed to be written in the case of servlet example to perform a basic job.

Besides HTML, a JSP may contain the following elements.

- **Directive Elements**
  - Provides global control of JSP .......................... <%@ %>

- **Scripting Elements**
  - JSP comments .............................................. <---%>
  - declarations .............................................. <%! %>
    - Used to declare instance variables & methods
  - expressions .............................................. <%= %>
    - A java code fragment which returns String
  - scriptlets .............................................. <% %>
    - Blocks of java code

- **Action Elements**
  - Special JSP tags ............................................. <jsp: ..... />

We’ll discuss in detail all the ingredients of JSP. This handout will cover only scripting elements, remaining ones will be discussed in next handouts.
Scripting Elements

Comments
Comments are ignored by JSP-to-servlet translator. Two types of comments are possibly used in JSP.

- **HTML comment:**

  These comments are shown in browser, means on taking view source of the web page; these sorts of comments can be read. Format of HTML comments is like:

  ```
  <!-- comment text -->
  ```

- **JSP comment:**

  These comments are not displayed in browser and have format like:

  ```
  <%= comment text %>
  ```

Expressions

The format of writing a Java expression is:  

```
<%= Java expression %>
```  

These expressions are evaluated, after converted to strings placed into HTML page at the place it occurred in JSP page. Examples of writing Expressions are:

- `<h2> Time: <%= new java.util.Date() %></h2>`
  will print current date & time after converting it to String

- `<h2> Welcome: <%= request.getParameter(“name”) %></h2>`
  will print the name attribute

Scriptlets

The format of writing a scriptlet is:  

```
<%= Java code %>
```  

After opening up the scriptlet tag, any kind of java code can be written inside it. This code is inserted verbatim into corresponding servlet.

Example of writing a scriptlet is:

```
<%  
    String n = request.getParameter(“name”);  
    out.println(“welcome ” + n);  
%>
```  

The above scriptlet reads the name attribute and prints it after appending “welcome”
Declarations

The format of writing a declaration tag is: `<%! Java code %>`

This tag is used to declare variables and methods at class level. The code written inside this tag is inserted verbatim into servlet’s class definition.

Example of declaring a class level (attribute) variable is:

```java
<%! 
private int someField = 5; %>
%
```

Example of declaring a class level method is:

```java
<%! 
public void someMethod ( ...... ) {

..............
}
%
```

Code Example: Using scripting elements

The next example code consists on two JSP pages namely first.jsp and second.jsp. The user will enter two numbers on the first.jsp and after pressing the calculate sum button, able to see the sum of entered numbers on second.jsp

**first.jsp**

This page only displays the two text fields to enter numbers along with a button.

```html
<html>
<body>
<h2>Enter two numbers to see their sum</h2>
<!—the form values will be posted to second.jsp -->
<form name="myForm" action="second.jsp">
<h3>First Number</h3>
<input type="text" name="num1" />
<h3>Second Number</h3>
<input type="text" name="num2" />
<input type="submit" value="Calculate Sum" />
</form>
</body>
</html>
```
second.jsp

This page retrieves the values posted by first.jsp. After converting the numbers into integers, displays their sum.

```html
<html>
<body>
<!-- JSP to sum two numbers -->
<%-- Declaration--%>
<%! // declaring a variable to store sum
    int res;
    // method helps in calculating the sum
    public int sum(int op1, int op2) {
        return op1 + op2;
    }
%>
<%-- Scriptlet--%>
<%
    String op1 = request.getParameter("num1");String op2 = request.getParameter("num2");
    int firstNum = Integer.parseInt(op1);
    int secondNum = Integer.parseInt(op2);
    // calling method sum(), declared above in declaration tag
    res = sum(firstNum, secondNum);
%>
<%-- expression used to display sum --%>
<h3>Sum is: <%=res%> </h3>
</body>
</html>
```

Writing JSP scripting Elements in XML

Now days, the preferred way for composing a JSP pages is using XML. Although writing JSP pages in old style is still heavily used as we had shown you in the last example. Equivalent XML tags for writing scripting elements are given below:

- **Comments**: No equivalent tag is defined
- **Declaration**: `<jsp:declaration> </jsp:declaration>`
- **Expression**: `<jsp:expression> </jsp:expression>`
- **Scriptlet**: `<jsp:scriptlet> </jsp:scriptlet>`

It’s important to note that every opening tag also have a closing tag too. The second.jsp of last example is given below in XML style.
<?xml version="1.0" encoding="UTF-8"?>

<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="2.0">

  <!-- to change the content type or response encoding change the following line-->
  <jsp:directive.page contentType="text/xml; charset=UTF-8"/>

  <!-- any content can be specified here, e.g.: -->
  <jsp:element name="text">
    <jsp:body>

      <jsp:declaration>
        int res;

        public int sum(int op1, int op2) {
          return op1 + op2;
        }
      </jsp:declaration>

      <jsp:scriptlet>
        String op1 = request.getParameter("num1");
        String op2 = request.getParameter("num2");
        int firstNum = Integer.parseInt(op1);
        int secondNum = Integer.parseInt(op2);
        res = sum(firstNum, secondNum);
      </jsp:scriptlet>

      <jsp:text>Sum is: </jsp:text>

      <jsp:expression>res</jsp:expression>

    </jsp:body>
  </jsp:element>
</jsp:root>

References

- Java A Lab Course by Umair Javed
- Core Servlets and JSP by Marty Hall
JavaServer Pages

We have started JSP journey in the last handout and thoroughly discussed the JSP scripting elements. JSP directive elements and implicit objects will be discussed in this handout. Let's review JSP journey again to find out what part we have already covered.

- **Directive Elements**
  - Provides global control of JSP .................................. <@ %>

- **Scripting Elements**
  - JSP comments .................................................. <----%>
  - declarations .................................................... <%! %>

- **Used to declare instance variables & methods**

  - expressions .................................................. <%= %>
    - A java code fragment which returns String
  - scriptlets .................................................... <% %>
    - Blocks of java code

  - A java code fragment which returns String

- **Action Elements**

  - Special JSP tags ............................................. <jsp: ..... />

We start our discussion from implicit objects. Let’s find out what these are?

**Implicit Objects**

To simplify code in JSP expressions and scriptlets, you are supplied with eight automatically defined variables, sometimes called implicit objects. The three most important variables are request, response & out. Details of these are given below:

- **request**

  This variable is of type HttpServletRequest, associated with the request. It gives you access to the request parameters, the request type (e.g. GET or POST), and the incoming HTTP request headers (e.g. cookies etc).

- **response**

  This variable is of type HttpServletResponse, associated with the response to client. By using it, you can set HTTP status codes, content type and response headers etc.
This is the object of JspWriter used to send output to the client.

**Code Example: Use of Implicit Objects**
The following example constitutes of 4 JSP pages. These are index.jsp, controller.jsp, web.jsp and java.jsp. The user will select either the option of “java” or “web” from index.jsp, displayed in the form of radio buttons and submits the request to controller.jsp. Based on the selection made by the user, controller.jsp will redirect the user to respective pages (web.jsp or java.jsp).

The flow of the example is shown below in the pictorial form.

The code of these entire pages is given below.

**index.jsp**

```html
<html>
<body>
<h2>Select the page you want to visit</h2>
<form name="myForm" action="controller.jsp">
<h3>
<input type="radio" name = "page" value="web"/>
Web Design & Develoment
</h3> <br>
<h3>
<input type="radio" name = "page" value="java"/>
</h3></form>
</body>
</html>
```
controller.jsp

<html>
<body>

<!-- scriptlet -->

<% 

// reading parameter “page”, name of radio button using
// implicit object request

String pageName = request.getParameter("page");

// deciding which page to move on based on “page” value
// redirecting user by using response implicit object

if (pageName.equals("web")) {
    response.sendRedirect("web.jsp");
} else if (pageName.equals("java")) {
    response.sendRedirect("java.jsp");
}

%>

</body>
</html>
web.jsp
<html>
<body>

// use of out implicit object, to generate HTML.

<% 
          out.println("<h2>" + 
                      "Welcome to Web Design & Development Page" + 
                      ");

%>

</body>
</html>

java.jsp
<html>
<body>

// use of out implicit object, to generate HTML.

<% 
          out.println("<h2>" + 
                      "Welcome to Java Page" + 
                      ");

%>

</body>
</html>
The details of remaining 5 implicit objects are given below:

- **session**
  This variable is of type HttpSession, used to work with session object.

- **application**
  This variable is of type ServletContext. Allows to store values in key-value pair form that are shared by all servlets in same web application.

- **config**
  This variable is of type ServletConfig. Represents the JSP configuration options e.g. init-parameters etc.

- **pageContext**
  This variable is of type javax.servlet.jsp.PageContext, to give a single point of access to many of the page attributes. This object is used to stores the object values associated with this object.

- **exception**
  This variable is of type java.lang.Throwable. Represents the exception that is passed to JSP error page.

- **page**
  This variable is of type java.lang.Object. It is synonym for this.
JSP Directives

JSP directives are used to convey special processing information about the page to JSP container. It affects the overall structure of the servlet that results from the JSP page. It enables programmer to:

- Specify page settings
- To Include content from other resources
- To specify custom-tag libraries

Format

<%@ directive {attribute=”val”}* %>

In JSP, there are three types of directives: page, include & taglib. The formats of using these are:

- **page**: `<%@ page {attribute=”val”}* %>`
- **include**: `<%@ include {attribute=”val”}* %>`
- **taglib**: `<%@ taglib {attribute=”val”}* %>`

**JSP page Directive**

Give high level information about servlet that will result from JSP page. It can be used anywhere in the document. It can control

- Which classes are imported
- What class the servlet extends
- What MIME type is generated
- How multithreading is handled
- If the participates in session
- Which page handles unexpected errors etc.

The lists of attributes that can be used with page directive are:

- **language** = “java”
- **extends** = “package.class”
- **import** = “package.*,package.class,…”
- **session** = “true | false”
- **Info** = “text”
- **contentType** = “mimeType”
- **isThreadSafe** = “true | false”
- **errorPage** = “relativeURL”
- **isErrorPage** = “true | false”

Some example uses are:

- To import package like java.util

  `<%@page import=“java.util.*” info=“using util package” %>`

- To declare this page as an error page

  `<%@ page isErrorPage = “true” %>`

- To generate the excel spread sheet

  `<%@ page contentType = “application/vnd.ms-excel” %>`
JSP include Directive

Let’s you include (reuse) navigation bars, tables and other elements in JSP page. You can include files at
– Translation Time (by using include directive)
– Request Time (by using Action elements, discussed in next handouts)

Format

<%@include file="relativeURL"%>

Purpose

To include a file in a JSP document at the time document is translated into a servlet. It may contain JSP code that affects the main page such as response page header settings etc.

Example Code: using include directive

This example contains three JSP pages. These are index.jsp, header.jsp & footer.jsp. The header.jsp will display the text of “web design and development” along with current date. The footer.jsp will display only “virtual university”. The outputs of both these pages will be included in index.jsp by using JSP include directive.

header.jsp

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

<html>

<body>

<marquee>

<h3>Web Desing & Development</h3>

<html><%!new Date()%></html>

</marquee>

</body>

</html>
footer.jsp

<html>
  <body>
    <marquee>
      <h3>Virtual University</h3>
    </marquee>
  </body>
</html>

index.jsp

<html>
  <body>
    // includes the output of header.jsp
    <%@include file="header.jsp" %>
    <table border=1>
      <tr><th>Apples</th><th>Oranges</th></tr>
      <tr><th>First Quarter</th><td>2307</td><td>4706</td></tr>
      <tr><th>Second Quarter</th><td>2982</td><td>5104</td></tr>
      <tr><th>Third Quarter</th><td>3011</td><td>5220</td></tr>
      <tr><th>Fourth Quarter</th><td>3055</td><td>5287</td></tr>
    </table>
    // includes the output of footer.jsp
    <%@include file="footer.jsp" %>
  </body>
</html>
Example Code: setting content type to generate excel spread sheet

In this example, index.jsp is modified to generate excel spread sheet of the last example. The change is shown in bold face.

**index.jsp**

// setting content type to generate excel sheet using page directive

```xml
<%@page contentType="application/vnd.ms-excel" %>
<html>
    <body>
        
            // includes the output of header.jsp
            <%@include file="header.jsp" %>
            
            <TABLE BORDER=1>
                <TR><TH><TH>Apples<TH>Oranges</TH></TR>
                <TR><TH>First Quarter</TH><TD>2307</TD><TD>4706</TD></TR>
                <TR><TH>Second Quarter</TH><TD>2982</TD><TD>5104</TD></TR>
                <TR><TH>Third Quarter</TH><TD>3011</TD><TD>5220</TD></TR>
                <TR><TH>Fourth Quarter</TH><TD>3055</TD><TD>5287</TD></TR>
            </TABLE>

            // includes the output of footer.jsp
            <%@include file="footer.jsp" %>
            
    </body>
</html>
```
JSP Life Cycle Methods
The life cycle methods of JSP are jspInit(), _jspService() and jspDestroy(). On receiving each request, _jspService() method is invoked that generates the response as well.

References:

- Java A Lab Course by Umair Javed
- Core Servlets and JSP by Marty Hall
Java Server Pages 3

In the last handout, we learned how to work with JSP directives and the use of implicit objects. In this handout, we'll learn about JavaBeans and what affect they produce. Before learning JavaBeans, let's start with an example that helps us to understand the impact of using JavaBeans.

Code Example: Displaying Course Outline

This example is actually the modification of the last one we had discussed in previous handout. User will select either course “web design and development” or “java”. On submitting request, course outline would be displayed of the selected course in tabular format. This course outline actually loaded from database. The schema of the database used for this example is given below:

The flow of this example is shown below:

index.jsp

This page is used to display the course options to the user in the radio button form.

<html>
<body>
<h2>Select the page you want to visit</h2>

<form name="myForm" action="controller.jsp" >
<h3> <input type="radio" name = "page" value="web"/> Web Design & Development</h3> <br>
</form>
</body>
</html>
controller.jsp
Based upon the selection made by the user, this page will redirect the user to respective pages. Those are web.jsp and java.jsp

```html
<html>
<body>

<!-- scriptlet -->

<!-- scriptlet -->

String pageName = request.getParameter("page");

// redirecting user based on selection made
if (pageName.equals("web"))
{
    response.sendRedirect("web.jsp");
}
else if (pageName.equals("java" ))
{
    response.sendRedirect("java.jsp");
}
%
```
web.jsp

This page is used to display course outline of “web design and development” in a tabular format after reading them from database. The code is:

// importing java.sql package using page directive, to work with database

<%@page import="java.sql.*%>
<html>
  <body>
    <center>
      <h2>Welcome to Web Design & Development Page</h2>
      <h3>Course Outline</h3>
      <table border="1">
        <tr>
          <th>Session No.</th>
          <th>Topics</th>
          <th>Assignments</th>
        </tr>
        <!-- start of scriptlet -->
        <% // establishing conection

          Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
          String conUrl = "jdbc:odbc:CourseDSN";
          Connection con = DriverManager.getConnection(conUrl);
        %>
      </table>
    </center>
  </body>
</html>
// preparing query using join statements

String sql = "SELECT sessionNo, topic, assignment " +" 
FROM Course, SessionDetail" +" WHERE courseName = ? " + " 
AND Course.courseId = SessionDetail.courseID";

PreparedStatement pStmt = con.prepareStatement(sql);

// setting parameter value "web".pStmt.setString(1, "web");
ResultSet rs = pStmt.executeQuery();

String sessionNo;
String topic;
String assignment;

// iterating over resultset 
while (rs.next()) {

    sessionNo = rs.getString("sessionNo");
    topic = rs.getString("topic");
    assignment = rs.getString("assignment");

    if (assignment == null) {
        assignment = "";
    }

}%>
<%-- end of scriptlet --%>

<%-- The values are displayed in tabular format using expressions, 
however it can also be done usingout.println(sessionNo) like statements 
--%>
<TR>
<TD><%=sessionNo%></TD>
<TD><%=topic%></TD>
<TD><%=assignment%></TD>
</TR>

<%}
} // end while
%

</TABLE>
</center>
</body>
</html>

**java.jsp**
The code of this page is very much alike of “web.jsp”. The only change is in making of query. Here the value is set “java” instead of “web”
// importing java.sql package using page directive, to work with database
<%@page import="java.sql.*%>

<html>
<body>
<center>

<h2>Welcome to Java Page </h2>
<h3>Course Outline</h3>

<TABLE BORDER="1">
<TR>
<TH>Session No.</TH>
<TH>Topics</TH>
<TH>Assignments</TH>
</TR>
<%!-- start of scriptlet --%>
<%
  // establishing connection
  Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
  String conUrl = "jdbc:odbc:CourseDSN";
  Connection con = DriverManager.getConnection(conUrl);

  // preparing query using join statements
  String sql = "SELECT sessionNo, topic, assignment " +" FROM Course, SessionDetail" +" WHERE courseName = ? " + " AND Course.courseId = SessionDetail.courseID";

  PreparedStatement pStmt = con.prepareStatement(sql);

  // setting parameter value "web".pStmt.setString(1, "java");

  ResultSet rs = pStmt.executeQuery();
  String sessionNo;
  String topic;
  String assignment;

  // iterating over resultset
  while (rs.next())
  {
    sessionNo = rs.getString("sessionNo");
    topic = rs.getString("topic");
    assignment = rs.getString("assignment");

    if (assignment == null)
{ 
    assignment = "";
}

<%-- end of scriptlet --%>

<%-- The values are displayed in tabular format using expressions, however it can also be done 
    using out.println(sessionNo) like statements
--%>

<TR>
    <TD> <%= sessionNo%> </TD>
    <TD> <%= topic%> </TD>
    <TD> <%= assignment%> </TD>
</TR>

<% } // end while
%

</TABLE>

</center>
</body>
</html>

Issues with Last Example
Too much cluttered code in web.jsp and java.jsp. This makes it very difficult to understand (probably you 
    experienced it by yourself) and to make changes/enhancements.
A single page is doing everything that is really a bad approach while making of web applications. The tasks 
    performed by web.jsp or java.jsp are:
    • Displaying contents (Presentation logic)
    • Connecting with database (DB connectivity logic)
    • Results Processing (Business Logic)

Can we simplify it? Yes, the answer lies in the use of JavaBeans technology.
JavaBeans
A java class that can be easily reused and composed together in an application. Any java class that follows certain design conventions can be a JavaBean.

JavaBeans Design Conventions
These conventions are:
• A bean class must have a zero argument constructor
• A bean class should not have any public instance variables/attributes (fields)
• Private values should be accessed through setters/getters
  For boolean data types, use boolean isXXX() & setXXX(boolean)
• A bean class must be serializable

A Sample JavaBean
The code snippet of very basic JavaBean is given below that satisfies all the conventions described above. The MyBean.java class has only one instance variable.

```java
public class MyBean implements Serializable {
    private String name; // zero argument constructor
    public MyBean() {
        name = "";
    }

    // standard setter
    public void setName(String n) {
        name = n;
    }

    // standard getter
    public String getName() {
        return name;
    }

    // any other method

    public void print() {
        System.out.println("Name is: "+name);
    }
}
```

} // end Bean class
Example Code: Displaying course outline by incorporating JavaBeans

This example is made by making more enhancements to the last one. Two JavaBeans are included in this example code. These are CourseOutlineBean& CourseDAO.
The CourseOutlineBean is used to represent one row of the table. It contains the following attributes:

- sessionNo
- topic
- assignment

The CourseDAO (where DAO stands of Data Acess Object) bean encapsulates database connectivity and result processing logic.
The web.jsp and java.jsp will use both these JavaBeans. The code of these and the JSPs used in this example are given below.

CourseOutlineBean.java
package vu;

import java.io.*;

public class CourseOutlineBean implements Serializable {
    private int sessionNo;
    private String topic;
    private String assignment;

    // no argument constructor
    public CourseOutlineBean() {
        sessionNo = 0;
        topic = "";
        assignment = "";
    }

    // setters
    public void setSessionNo(int s) {
        sessionNo = s;
    }

    public void setTopic(String t) {
        topic = t;
    }
}
public void setAssignment(String a) {
    assignment = a;
}

// getters
public int getSessionNo() {
    return sessionNo;
}

public String getTopic() {
    return topic;
}

public String getAssignment() {
    return assignment;
}

} // end class

CourseDAO.java

package vu;

import java.io.*;
import java.sql.*;
import java.util.*;

public class CourseDAO implements Serializable {
    private Connection con;

    public CourseDAO() {
        establishConnection();
    }

    private void establishConnection() {
        // establish connection logic
    }

    // other methods
}

// Additional methods and logic
```java
// establishConnection method
void establishConnection() {
    try {
        // establishing connection
        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
        String conUrl = "jdbc:odbc:CourseDSN";
        con = DriverManager.getConnection(conUrl);
    } catch (Exception ex) {
        System.out.println(ex);
    }
}

// retrieveCourseList method
public ArrayList retrieveCourseList(String cName) {
    ArrayList courseList = new ArrayList();
    try {
        String sql = "SELECT sessionNo, topic, assignment " + "";
        FROM Course, SessionDetail" + "";
        WHERE courseName = ? " + "";
        AND Course.courseId = SessionDetail.courseID ";

        PreparedStatement pStmt = con.prepareStatement(sql);
        pStmt.setString(1, cName);
        ResultSet rs = pStmt.executeQuery();
    }
```
int sNo;
String topic;
String assignment;

while ( rs.next() ) {
    sNo = rs.getInt("sessionNo");
    topic = rs.getString("topic");
    assignment = rs.getString("assignment");

    if (assignment == null)
    {
        assignment = "";
    }

    // creating a CourseOutlineBean object
    CourseOutlineBean cBean = new CourseOutlineBean();
    cBean.setSessionNo(sNo);
    cBean.setTopic(topic);
    cBean.setAssignment(assignment);

    // adding a bean to arraylist
    courseList.add(cBean);
}
} catch(Exception ex){
    System.out.println(ex);
} finally {
    // to close connection
    releaseResources();
}

// returning ArrayList object
return courseList;
} // end retrieveCourseOutline
/** ********** releaseResources method ***************

private void releaseResources() {
    try {
        if (con != null) {
            con.close();
        }
    } catch (Exception ex) {
        System.out.println();
    }
}

} // end releaseResources

} // end CourseDAO

index.jsp

This page is used to display the course options to the user in the radio button form.

<html>
<body>
<h2>Select the page you want to visit</h2>

<form name="myForm" action="controller.jsp">
    <h3><input type="radio" name = "page" value="web" />
        Web Design & Development</h3>

    <h3><input type="radio" name = "page" value="java" />
        Java</h3>

    <input type="submit" value="Submit" />
</form>
</body>
</html>
controller.jsp

Based on user selection, redirects the user to desired page.

```html
<html>
<body>
<!-- scriptlet -->
<%
    String pageName = request.getParameter("page");
    if (pageName.equals("web")) {
        response.sendRedirect("web.jsp");
    } else if (pageName.equals("java")) {
        response.sendRedirect("java.jsp");
    }
%>
</body>
</html>
```

web.jsp

This page is used to display course outline of “web design and development” in a tabular format after reading them from database. Moreover, this page also uses the JavaBeans (CourseOutlineBean & CourseDAO).

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

<html>
<body>
<center>
```
Welcome to Web Design & Development Course

Course Outline

<table>
<thead>
<tr>
<th>Session No.</th>
<th>Topics</th>
<th>Assignments</th>
</tr>
</thead>
</table>

-- start of scriptlet --

// creating CourseDAO object
CourseDAO courseDAO = new CourseDAO();

// calling retrieveCourseList() of CourseDAO class and
// passing “web” as value. This method returns ArrayList

courseList = courseDAO.retrieveCourseList("web");

CourseOutlineBean webBean = null;

// iterating over ArrayList to display course outline
for(int i=0; i<courseList.size(); i++){
    webBean = (CourseOutlineBean)courseList.get(i);
}

-- end of scriptlet --


Web Design & Development – CS506

java.jsp
The code contains by this page is almost same of web.jsp. Here, “java” is passed to retrieveCourseList() method. This is shown in boldface.

<%@ page import="java.util.*" %>
<%@ -- importing vu package that contains the JavaBeans--%><%@ page import="vu.*" %>

<html>
<body>
<center>
<h2>Welcome to Java Course </h2>
<h3>Course Outline</h3>

<TABLE BORDER="1">
<TR>
  <TH>Session No.</TH>
  <TH>Topics</TH>
  <TH>Assignments</TH>
</TR>
</html>
<%-- start of scriptlet --%>
<%   
   // creating CourseDAO object
   CourseDAO courseDAO = new CourseDAO();
     
   // calling retrieveCourseList() of CourseDAO class and
   // passing “java” as value. This method returns ArrayList
   ArrayList courseList = courseDAO.retrieveCourseList("java");
   CourseOutlineBean javaBean = null;

   // iterating over ArrayList to display course outline
   for(int i=0; i<courseList.size(); i++) {
      javaBean = (CourseOutlineBean) courseList.get(i);
   %>

   <%%-- end of scriptlet --%>

   <TR>
      <TD><%= javaBean.getSessionNo()%></TD>
      <TD><%= javaBean.getTopic()%></TD>
      <TD><%= javaBean.getAssignment()%></TD>
   </TR>

   <%
   %>
   } // end for

</TABLE>
References:
Entire material for this handout is taken from the book JAVA A Lab Course by Umair Javed. This material is available just for the use of VU students of the course Web Design and Development and not for any other commercial purpose without the consent of author.
JSP Action Elements and Scope

The journey we had started of JSP is very much covered except of JSP action elements. In this handout, we’ll study the use of JSP action elements. Further also learn how and where to store JavaBean objects that can be shared among JSP pages.

Let’s first quickly look on the JSP journey to find out where we have reached.

- **Directive Elements**
  - Provides global control of JSP ..........................................<%@ %>

- **Scripting Elements**
  - JSP comments .........................................................<%---%>
  - declarations ..........................................................<%=%>

- **Used to declare instance variables & methods**

  ```
  implicit_objects
  \{
  \{expressions \.................................................... <%= %
  \}\n  \{scriptlets \.................................................... <% %
  \}\n  ```

- **Action Elements**
  - Special JSP tags .................................................... <jsp: ...... />

JSP Action Elements

JSP action elements allow us to work with JavaBeans, to include pages at request time and to forward requests to other resources etc.

**Format**

Expressed using XML syntax

- Opening tag `<jsp:element attribute="value" ..... >`
- Body `body`
- Closing tag `<jsp:element>`

Empty tags (without body) can also be used like

`<jsp:element attribute="value" ..... >`

Some JSP Action Elements

- To work with JavaBeans
  - `<jsp:useBean />
  - `<jsp:setProperty />
  - `<jsp:getProperty />

- To include resources at request time
  - `<jsp:include />

- To forward request to another JSP or Servlet
Working with JavaBeans using JSP Action Elements

The three action elements are used to work with JavaBeans. These are discussed in detail below.

JSP useBean Action Element

It is used to obtain a reference to an existing JavaBean object by specifying id(name of object) and scope in which bean is stored. If a reference is not found, the bean is instantiated.

The format of this action element is:

```jsp
<jsp:useBean id = “name” scope = “page | request | session | application”
    class = “package.Class” />
```

The id attribute specifies the name of the JavaBean object that is also used for later references. The scope attribute can have one possible value out of page, request, session and application. If this attribute is omitted, the default value of scope attribute is page. We’ll discuss in detail about scope shortly.

The class attribute specifies the type of object is going to be created.

`jsp:useBean` is being equivalent to building an object in scriptlet. For example to build an object of MyBean using scriptlet is:

```jsp
<% MyBean m = new MyBean(); %>
```

Achieving above functionality using `jsp:useBean` action element will look like this:

```jsp
<jsp:useBean id = “m” scope = “page” class = “vu.MyBean” />
```

In the above code snippet, we are assuming that MyBean lies in vu package.

JSP setProperty Action Element

To set or change the property value of the specified bean. String values are converted to types of properties by using the related conversion methods.

The format of this action element is:

```jsp
<jsp:setProperty name = “beanName or id” property = “name” value = “value” />
```

The name attribute should match the id given in `jsp:useBean`. The property attribute specifies the name of the property to change and the value attribute specifies the new value.

`jsp:setProperty` is being equivalent to following code of scriptlet. For example to change the name property of m (instance of MyBean) using scriptlet is:

```jsp
<% 
    m.setProperty(“ali”);
%>
```

Achieving above functionality using `jsp:setProperty` action element will look like this:

```jsp
<jsp:setProperty name = “m” property = “name” value = “ali” />
```
JSP `getProperty` Action Element

Use to retrieves the value of property, converts it to String and writes it to output stream. The format of this action element is:

```jsp
<jsp:getProperty name = “beanName or id” property = “name” />
```

`getProperty` is being equivalent to following code of scriptlet. For example to retrieve the name property of m (instance of MyBean) followed by writing it to output stream, scriptlet code will look like:

```jsp
<%
    String name = m.getName();
    out.println(name);
%
```

Achieving above functionality using `getProperty` action element will look like this:

```jsp
<jsp:getProperty name = “m” property = “name” />
```

Example Code: Calculating sum of two numbers by using action elements and JavaBean

This example contains index.jsp and result.jsp and one JavaBean i.e. SumBean. User will enter two numbers on index.jsp and their sum will be displayed on result.jsp. Let’s examine these one after another

**SumBean.java**

The SumBean has following attributes

- `firstNumber`
- `secondNumber`
- `sum`

The `firstNumber` and `secondNumbers` are “write-only” properties means for these only setters would be defined. Whereas sum is a “read-only” property as only getter would be defined for it.

The SumBean also contain one additional method for calculating sum i.e. `calculateSum()`. After performing addition of `firstNumber` with `secondNumber`, this method will assign the result to sum attribute.

```java
package vu;
import java.io.*;
public class SumBean implements Serializable{
    private int firstNumber;
    private int secondNumber;
    private int sum;

    // no argument constructor
    public SumBean() {
        firstNumber = 0;
        secondNumber = 0;
        sum = 0;
    }
}
```
// firstNumber & secondNumber are writeonly properties
public void setFirstNumber(int n) {
    firstNumber = n;
}

public void setSecondNumber(int n) {
    secondNumber = n;
}

// no setter for sum
// sum is a read only property
public int getSum() {
    return sum;
}

// method to calculate sum
public void calculateSum() {
    sum = firstNumber + secondNumber;
}

index.jsp

This page will display two text fields to enter number into them.

<html>
<body>

<h2>Enter two numbers to calculate their sum</h2>
<form name="myForm" action="result.jsp">

<h3>Enter first number <input type="text" name="num1" /></h3>
    Enter second number
    <input type="text" name="num2" />
    <br/>
    <input type="submit" value="Calculate Sum" />
</form>

</body>
</html>
result.jsp
This page will calculate the sum of two entered numbers by the user and displays the sum back to user. The addition is performed using SumBean

<%@page import="vu.*"%> <html> <body>
<h2>The sum is: </h2>
<%-- importing vu package that contains the SumBean --%>

<%-- instantiating bean using action element --%>
    //Servlet equivalent code of useBean
    SumBean sBean = new SumBean();
    --%>

  <jsp:useBean id="sBean" class="vu.SumBean" scope="page"/>
  <%-- setting firstNumber property of sBean using action elements --%>
  --%>

  <jsp:setProperty name="sBean" property="firstNumber" param="num1" />
  <%-- Servlet equivalent code of setProperty for num1 int no =
  Integer.parseInt(request.getParameter("num1"));sBean.setFirstNumber(no);
  --%>

  <jsp:setProperty name="sBean" property="secondNumber" param="num2" />
  <%-- Servlet equivalent code of setProperty for num2 int no =
  Integer.parseInt(request.getParameter("num2"));sBean.setSecondNumber(no);
  --%>

  <jsp:getProperty name="sBean" property="sum" />
</h2>

</body>
</html>
Sharing Beans & Object Scopes
So far, we have learned the following techniques to create objects.
- Implicitly through JSP directives
- Explicitly through actions
- Directly using scripting code

Although the beans are indeed bound to local variables, that is not the only behavior. They are also stored in four different locations, depending on the value of the optional scope attribute of jsp:useBean. The scope attribute has the following possible values: page, request, session and application.

Let’s discover what impact these scopes can produce on JavaBeans objects which are stored in one of these scopes.

- page

This is the default value of scope attribute, if omitted. It indicates, in addition to being bound to local variable, the bean object should be placed in the pageContext. The bean’s values are only available and persist on JSP in which bean is created.

In practice, beans created with page scope are always accessed (their values) by jsp:getProperty, jsp:setProperty, scriptlets or expressions later in the same page. This will be more cleared with the help of following diagram:

In the diagram above, first.jsp generates a request “request 1” that is submitted to second.jsp. Now, second.jsp creates an object m of MyBean by calling its default constructor and stores a value “ali” for the name property by making a call to appropriate setter method. Since, the scope specified in this example is “page” when the object of MyBean is instantiated using jsp:useBean action element. Therefore, object (m) of MyBeans is stored in PageContext.

Whether, second.jsp forwards the same request (request 1) to third.jsp or generates a new request (request 2), at third.jsp, values (e.g. ali) stored in MyBean object m, are not available. Hence, specifying scope “page” results in using the object on the same page where they are created.
- **request**

  This value signifies that, in addition to being bound to local variable, the bean object should be placed in ServletRequest object for the duration of the current request. In other words, until you continue to forward the request to another JSP/servlet, the beans values are available. This has been illustrated in the following diagram.

  ![Diagram](image)

  In the diagram above, MyBean is instantiated by specifying scope = “request” that results in storing object in ServletRequest. A value “ali” is also stored in m using setter method.

  **second.jsp** forwards the same request (request 1) to third.jsp, since scope of m (object of MyBean) is request, as a result third.jsp can access the values(e.g. ali) stored in m. According to the figure, third.jsp generates a new request (request 2) and submits it to fourth.jsp. Since a new request is generated therefore values stored in object m (e.g. ali) are not available to fourth.jsp.

- **session**

  This value means that, in addition to being bound to local variable, the bean object will be stored in the HttpSession object associated with the current request. As you already know, object’s value stored in HttpSession persists for whole user’s session. The figure below helps in understanding this concept.
In the diagram above, MyBean is instantiated by specifying scope = “session” that results in storing object in HttpSession. A value “ali” is also stored in m using setter method. Irrespective of request forwarding or new request generation from second.jsp to other resources, the values stored in HttpSession remains available until user’s session is ended.

- application

This very useful value means that, in addition to being bound to local variable, the bean object will be stored in ServletContext. The bean objects stored in ServletContext is shared by all JSPs/servlets in the same web application. The diagram given below illustrates this scenario:
Summary of Object's Scopes

Let's take another view of session, request & page scopes in the next figure that helps us to understand the under beneath things.
The figure shows four JavaServer Pages. Each page has its own page scope. Therefore objects stored in page scope are only available to same pages on which they are created. Suppose page1 forwards the request to page2. Objects stored in request scope remains available to page1 as well to page 2. Similar case is true for page 3 & page 4. If user makes a visit to all these pages in one session, object’s values stored in session scope remains available on all these pages.

To understand the difference between sessions & application scope, consider the following figure:

As you can conclude from the figure, for each user (client), objects are stored in different sessions. However, in the case of application scope, all users stores objects in single place.
More JSP Action Elements
Let’s talk about two important action elements. These are include & forward.

JSP include action Element
It is used to include files at request time. For example, to reuse HTML, JSP or plain text content. It’s important to note that JSP content cannot affect main page (in which output is included); only output of included JSP is used. It also allows updating of the included content without changing the main JSP.

The jsp:include action element requires two attributes: page & flush. -page: a relative URL of the file to be included. -flush: must have the value “true”

```<jsp:include page="relative URL" flush="true" />
```

jsp:include is being equivalent to following code of scriptlet. For example to include the output of one.jsp, scriptlet code will look like:

```<% RequestDispatcher rd = request.getRequestDispatcher("one.jsp"); rd.include(request, response);
%
```

Achieving above functionality using jsp:include action element will look like this:

```<jsp:include page="one.jsp" flush="true" />
```

JSP forward action Element
It is used to forward request to another resource. The format of jsp:forward action is:

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

jsp:forward is being equivalent to following code of scriptlet. For example to forward the request to one.jsp, scriptlet code will look like:

```<% RequestDispatcher rd = request.getRequestDispatcher("one.jsp"); rd.forward(request, response);
%
```

References:
- Java A Lab Course by Umair Javed.
- Core Servlets and JavaServer Pages by Marty Hall
JSP Custom Tags

To begin with, let’s review our last code example of lecture 36 i.e. Displaying course outline. We incorporated JavaBeans to minimize the database logic from the JSP. But still, we have to write some lines of java code inside java.jsp & web.jsp. As discussed earlier, JSPs are built for presentation purpose only, so all the other code that involves business and database logic must be shifted else where like we used JavaBeans for such purpose.

There is also another problem attached to it. Generally web page designers which have enough knowledge to work with HTML and some scripting language, faced lot of difficulties in writing some simple lines of java code. To overcome these issues, java provides us the mechanism of custom tags.

Motivation

To give you an inspiration, first have a glance over the code snippet we used in JSP of the course outline example of last lecture. Of course, not all code is given here; it’s just for your reference to give you a hint.

```jsp
<%  
    CourseDAO courseDAO = new CourseDAO();
    .................
// iterating over ArrayList
for (..................... ) {
    ....................
    ....................
// displaying course outline
}
%>
```

Can we replace all the above code with one single line? Yes, by using custom tag we can write like this:

```jsp
<mytag:course tag pageName=“java” />
```

By only specifying the course/page name, this tag will display the course outline in tabular format. Now, you must have realized how significant changes custom tags can bring on.

What is a Custom Tag?

- In simplistic terms, “a user defined component that is used to perform certain action”. This action could be as simple as displaying “hello world” or it can be as complex as displaying course outline of selected course after reading it from database.
- It provides mechanism for encapsulating complex functionality for use in JSPs. Thus facilitates the non-java coders.
• We already seen & used many built in tags like:
  - &lt; jsp:useBean …… /&gt;
  - &lt; jsp:include …… /&gt;
  - &lt; jsp:forward …… /&gt; etc.

Why Build Custom Tag?

• We introduced action &lt;jsp:useBean&gt; and JavaBeans to incorporate complex, encapsulated
  functionality in a JSP.
• However, JavaBeans cannot manipulate JSP content and Web page designers must have some
  knowledge to use JavaBeans in a page
• With Custom tags, it is possible for web page designers to use complex functionality without knowing
  any java

Advantages of using Custom Tags

• Provides cleaner separation of processing logic and presentation, than JavaBeans.
• Have access to all JSP implicit objects like out, request etc.
• Can be customized by specifying attributes.

Types of Tags

Three types of can be constructed. These are:

1 Simple Tag
2 Tag with Attribute
3 Tag with Body

1. Simple Tag

A simple tag has the following characteristics:
  - Start and End of tag - No body is specified within tag - No attributes - For example

```html
&lt; mytag:hello /&gt
```

Tag Library Prefix Tag Name
2. Tag with Attributes

A tag with attributes has the following characteristics:
- Start and End of tag
- Attributes within tag
- No body enclosed
- For example
  
  `<mytag:hello attribute = “value” />`

3. Tag with Body

A tag with body has the following characteristics:
- Start and End of tag
- May be attributes
- Body enclosed within tag
- For example
  
  `<mytag:hello optional_attributes ………… >

  some body

  </mytag:hello>`
Building Custom Tags

So far, we have used many built-in tags. Now the time has come to build your own one. Custom tags can be built either by using JSP 1.2 specification or JSP 2.0 (latest) specification.

To develop custom tags using JSP 1.2 involves lot of cumbersome (too difficult for James Gossling also☺). However, JSP 2.0 brings lots of goodies like

- Simple tag extensions to build custom tags
- Integrated Expression Language (will be discussed in coming lecture)
- Also provides an alternate mechanism for building custom tags using tag files (.tag)
- Improved XML syntax etc.

Steps for Building Custom Tags

The following steps are used in order to develop your own custom tag. These are:

1. Develop the Tag Handler class

   - Tag Handler is also a java class that is implicitly called when the associated tag is encountered in the JSP.
   - Must implement SimpleTaginterface
   - Usually extend from SimpleTagSupport class that has already implemented SimpleTag interface.
   - For example,

     ```java
     public class MyTagHandler extends SimpleTagSupport {
         ......................
         ......................
     }
     ```

2. doTag() method

   - By default does nothing
   - Need to implement / override to code/write functionality of tag
   - Invoked when the end element of the tag encountered.

   - JSP implicit objects (e.g. out etc) are available to tag handler class through pageContextobject.
   - pageContextobject can be obtained using getJspContext() method.
   - For example to get the reference of implicit outobject, we write.

     ```java
     PageContext pc = (PageContext) getJspContext();
     JspWriter out = pc.getOut();
     ```

3. Write Tag Library Descriptor (.tld) file

   - It is a XML based document.
   - Specifies information required by the JSP container such as:
- Tag library version
- JSP version
- Tag name
- Tag Handler class name
- Attribute names etc.

**Note:** If you are using any IDE (like netBeans® 4.1, in order to build custom tags, the IDE will write .tldfile for you.

3. Deployment

- Place Tag Handler class in myapp/WEB-INF/classes folder of web application.
- Place .tld file in myapp/WEB-INF/tlds folder of web application.

**Note:** Any good IDE will also perform this step on your behalf Use taglib directive in JSP to refer to the tag library. For example

```<%@ taglib uri=""TLD file name"" prefix="mytag" %>```

The next step is to call the tag by its name as defined in TLD. For example, if tag name is hello then we write:

```< mytag:hello />```  
where mytag is the name of prefix specified in taglibdirective.

What actually happened behind the scenes? Container calls the doTag() method of appropriate tag handler class. After that, Tag Handler will write the appropriate response back to the page.

**Example Code: Building simple tag that displays “Hello World”**

Enough we have talked about what are custom tags, their types. Now, it is a time to build a custom tag that displays “Hello World”.

**Approach**

- Extend Tag Handler class from SimpleTagSupport class and override doTag() method
- Build TLD file
- Deploy

**Note:** As mentioned earlier, if you are using any IDE (like netBeans® 4.1), the last two steps will be performed by the IDE.

**WelcomeTagHandler.java**

```java
package vu;
// importing required packages
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
```
// overriding doTag() method
public void doTag() throws JspException {

   // obtaining the reference of out implicit object
   PageContext pageContext = (PageContext)getJspContext(); JspWriter out = pageContext.getOut();
   try {
      out.println("Hello World ");

      } catch (java.io.IOException ex) {throw new
            JspException(ex.getMessage());}

   } // end doTag() method
}
}

WelcomeTagHandler class

customtags.tld

If using IDE, this file will be written automatically. In this file you specify the tag name along with Tag
Handler class.

<?xml version="1.0" encoding="UTF-8"?>
<taglib version="2.0"
   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">
   <tlib-version>1.0</tlib-version>
   <short-name>mytag</short-name>
   <!--the value of uri will be used in JSP to refer to this tld -->
   <uri>/WEB-INF/tlds/customtags</uri>

   <!— Specifying the tag name and tag class. Also mentioning that this tag has no body
   -->
   <tag>
      <name>welcome</name>
      <tag-class>vu.WelcomeTagHandler</tag-class>
      <body-content>empty</body-content>
   </tag>

</taglib>
using taglib directive, specifying the tld file name as well as prefix. Note that you use any value for the prefix attribute --%>

<%@taglib uri="/WEB-INF/tlds/customtags.tld" prefix="mytag" %>

<html> <body>

<h2>A Simple Tag Example</h2>

<h3><%-- calling welcome tag with the help of prefix --%><mytag:welcome /> </h3>

</body></html>

Building tags with attributes

If you want to build a tag that can also take attributes, for example

<mytag:hello attribute="value" />

To handle attributes, you need to add
- Instance variables and
- Corresponding setter methods

Behind the scenes, container will call these setter methods implicitly and pass the value of the custom tag attribute as an argument.

In this example, we will modify our course outline example to incorporate tags. Based on attribute value, the tag will display the respective course outline in tabular format.

Approach
- Extend Tag Handler class from SimpleTagSupport class
  - Add instance variable of type String
  - Write setter method for this attribute
  - Override doTag() method

- Build TLD file
- Deploy
CourseOutlineBean.java
This is the same file used in the last example
package vubean;

import java.io.*;

public class CourseOutlineBean implements Serializable {

    private int sessionNo;
    private String topic;
    private String assignment;

    // no argument constructor
    public CourseOutlineBean() {
        sessionNo = 0;
        topic = "";
        assignment = "";
    }

    // setters
    public void setSessionNo(int s) {
        sessionNo = s;
    }

    public void setTopic(String t) {
        topic = t;
    }

    public void setAssignment(String a) {
        assignment = a;
    }

    // getters
    public int getSessionNo() {return sessionNo;}

    public String getTopic() {return topic;}

    public String getAssignment() {return assignment;}

} // end class
CourseDAO.java
No changes are made to this file too.

package vu;

import java.io.*; import java.sql.*; import java.util.*;

public class CourseDAO implements Serializable{

    private Connection con;

    public CourseDAO() {
        establishConnection();
    }

    //********** establishConnection method **********
    // method used to make connection with database
    private void establishConnection() {
        try { // establishing connexionClass.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            String conUrl = "jdbc:odbc:CourseDSN"; con =
                DriverManager.getConnection(conUrl);
        } catch (Exception ex) {
            System.out.println(ex);
        }
    }

    ArrayList courseList = new ArrayList();
    try {
        String sql = "SELECT sessionNo, topic, assignment " +" FROM Course, SessionDetail" +"
            WHERE courseName = ? " + " AND Course.courseId = 
            SessionDetail.courseID ";

        PreparedStatement pStmt = con.prepareStatement(sql); pStmt.setString(1, cName);

        ResultSet rs = pStmt.executeQuery();

        int sNo;
        String topic;
        String assignment;
while ( rs.next() ) {
    sNo = rs.getInt("sessionNo");
    topic = rs.getString("topic");
    assignment = rs.getString("assignment");
    if (assignment == null) { assignment = ""; }

    // creating a CourseOutlineBean object
    CourseOutlineBean cBean = new CourseOutlineBean();
    cBean.setSessionNo(sNo);
    cBean.setTopic(topic);
    cBean.setAssignment(assignment);

    // adding a bean to arraylist
    courseList.add(cBean);
}

} catch(Exception ex) {
    System.out.println(ex);
}

} finally {
    // to close connection
    releaseResources();
}

} // end retrieveCourseOutline

} // end CourseDAO
**MyTagHandler.java**

The tag handler class uses JavaBeans (CourseOutlineBean.java & CourseDAO.java), and includes the logic of displaying course outline in tabular format.

```java
package vutag;

// importing package that contains the JavaBeansimport vubean.*;

import javax.servlet.jsp.tagext.*;import
javax.servlet.jsp.*;import java.util.*;

public class MyTagHandler extends SimpleTagSupport {

    /**
     * Declaration of pageName property.
     */
    private String pageName;

    public void doTag() throws JspException {
        CourseDAO courseDAO = new CourseDAO();
        ArrayList courseList = courseDAO.retrieveCourseList(pageName);

        // to display course outline in tabular form, this method is// used – define below
display(courseList);
    }

    /**
     * Setter for the pageName attribute.
     */
    public void setPageName(java.lang.String value) {this.pageName = value;
    }

    /**
     * display method used to print courseoutline in tabular form
     */private void display(ArrayList courseList) throws JspException{
        PageContext pc = (PageContext)getJspContext();
        JspWriter out = pc.getOutputStream();

        try{
```
// displaying table headers

out.print("<TABLE BORDER=1">");
out.print("<TR>");
out.print("<TH> Session No </TH>");
out.print("<TH> Topic </TH>");
out.print("<TH> Assignment </TH>");
out.print("<TR>");

// loop to iterate over courseList
for (int i=0; i<courseList.size(); i++){
    CourseOutlineBean courseBean = (CourseOutlineBean)courseList.get(i);

    // displaying one row
    out.print("<TR>");
    out.print("<TD>") + courseBean.getSessionNo() + "</TD>");
    out.print("<TD>") + courseBean.getTopic() + "</TD>");
    out.print("<TD>") + courseBean.getAssignment() + "</TD>");
    out.print("<TR>");
}
out.print("</TABLE>");

} // catch(java.io.IOException ex) {throw new JspException(ex.getMessage());}
// end clas MyTagHandler.java

<?xml version="1.0" encoding="UTF-8"?>
<taglib version="2.0"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee web-jsptaglibrary_2_0.xsd"
    <tlib-version>1.0</tlib-version>
    <short-name>mytaglibrary</short-name>
    <!--the value of uri will be used in JSP to refer to this tld -->
    <uri>/WEB-INF/tds/mytaglibrary</uri>

    <!-- Specifying the tag name and tag class. Also mentioning that this tag has no body -->
    <tag>
        <name>courseTag</name>
        <tag-class>vutag.MyTagHandler</tag-class>
        <body-content>empty</body-content>

index.jsp

This page is used to display the course options to the user in the radio button form.

<html>
  <body>
    <h2>Select the page you want to visit</h2>
    <form name="myForm" action="controller.jsp">
      <h3><input type="radio" name = "page" value="web"/>Web Design & Development</h3>
      <br/>
      <h3><input type="radio" name = "page" value="java"/>Java</h3>
      <br/>
      <input type="submit" value="Submit"/>
    </form>
  </body>
</html>
controller.jsp
Based upon the selection made by the user, this page will redirect the user to respective pages. Those are web.jsp and java.jsp

<html>
  <body>
  <!-- scriptlet -->
  <%
    String pageName = request.getParameter("page");
    if (pageName.equals("web")) {
      response.sendRedirect("web.jsp");
    } else if (pageName.equals("java")) {
      response.sendRedirect("java.jsp");
    }
  %>
  </body>
</html>

java.jsp

<%@taglib uri="/WEB-INF/tlds/mytaglibrary.tld" prefix="mytag"%>

<html>
  <body>
    <center>
      <h2>Welcome to Java Learning Center</h2>
      <h3>Course Outline</h3>
      <!--calling coursetag and specifying java as attribute value
      -->
      <mytag:coursetag pageName="java"/>
    </center>
  </body>
</html>
web.jsp

<%@taglib uri="/WEB-INF/tlds/mytaglibrary.tld" prefix="mytag"%>

<html>
<body>
  <center>
    <h2>Welcome to Java Learning Center</h2>
    <h3>Course Outline</h3>
  </center>
  <%--calling coursetag and specifying java as attribute value--%>
  <mytag:coursetag pageName="java" />
</body>
</html>

References

- Java A Lab Course by Umair Javed.
- Core Servlets and JavaServer Pages by Marty Hall
MVC + Case Study

We have covered an adequate amount of Servlets and JSPs in detail. Now, the time has come to learn different architectures that are most commonly used for the sake of web development. These architectures also help us to understand where these components best fit in. In this handout, we'll cover the most widely used/popular architecture i.e. Model View Controller (MVC).

A small case study “Address Book” is also part of this handout that is based on MVC Model 1. Before moving on to MVC, let's see what error pages are and how they are used?

Error Page

Error Pages enables you to customize error messages. You can even hide them from the user's view entirely, if you want. This also makes possible to maintain a consistent look and feel throughout an application, even when those dreaded error messages are thrown.

By means of page directive, a JSP can be given the responsibility of an Error page. An Error JSP is called by the web server when an uncaught exception gets occurred. This exception is passed as an instance of java.lang.Throwable to Error JSP (also accessible via implicit exception object).

Defining and Using Error Pages

- isErrorPage attribute of a page directive is used to declare a JSP as an error page.
- JSP pages are informed about the error page by setting errorPage attribute of page directive

In the figure below, error.jsp is defined as JSP Error page and index.jsp is informed to call error.jsp if any uncaught exception rose. This is done by setting attributes errorPage and isErrorPage of the page directive on these JSPs.
Case Study – Address Book

What we have learned is going to be implemented in this Address Book example. Here MS-Access is being used as DBMS. This database will have only one table, Person with following attributes

Ingredients of Address Book

Java Beans, Java Server Pages and Error Page that are being used in this Address Book example are: -

. Java Beans
  . PersonInfo – Has following attributes:
    – name
    – address
    – phoneNumber

  . PersonDAO
    – Encapsulates database logic.
    – Therefore, it will be used to save and retrieve PersonInfo data.

. Java Server Pages
  . addperson.jsp
    – Used to collect new person info that will be saved in database.

  . saveperson.jsp
    – Receives person info from addperson.jsp
    – Saves it to database

  . searchperson.jsp
    – Used to provide search criteria to search Person’s info by providing name
. **showperson.jsp**

  – This page receive person’s name from searchperson.jsp to search in database
  – Retrieves and displays person record found against person name

**Error Page**

. **addbookerror.jsp**

  – This page is declared as an error page and used to identify the type of exception.
  – In addition to that, it also displays the message associated with the received exception to the user.

**Program Flow**

Now let’s discuss the flow of program. Assume that the system has been deployed on a JSP compatible Web Server like Tomcat and has been ready to use for clients. The following figure helps to understand the program flow of this small example.

```
addperson.jsp  +-----------------+  saveperson.jsp  
               |  uses             |              +-----------------+
               v  JavaBeans  
searchperson.jsp  +-----------------+  showperson.jsp  
          +-----------------+  exception  
          v  uses
addbookerror.jsp
```

addperson.jsp takes person’s information from the user and sends it to saveperson.jsp. After receiving request, saveperson.jsp makes an object of PersonInfo using received information and saves it into the database using PersonDAOJava bean.

Similarly, searchperson.jsp takes search criteria (name) from the user and passes it to showperson.jsp that searches the record in database using PersonDAO and shows the results to the user.

If any uncaught exception is generated on these JSP, addbookerror.jsp is called implicitly, which displays an appropriate message to the user after identifying the exception type.
Code for the Case Study

Let’s have a look on the code of each component used in the case study; first start from JavaBeans.

**PersonInfo**

PersonInfo represents the record of one person and its objects are used to interrupt the information about persons.

```java
package vu;

import java.io.*;

public class PersonInfo implements Serializable {

    private String name;
    private String address;
    private int phoneNum;

    // no argument constructor
    public PersonInfo() {
        name = "";
        address = "";
        phoneNum = 0;
    }

    // setters
    public void setName(String n) {
        name = n;
    }

    public void setAddress(String a) {
        address = a;
    }

    public void setPhoneNum(int pNo) {
        phoneNum = pNo;
    }

    // getters
    public String getName() {
        return name;
    }

    public String getAddress() {
        return address;
    }
}
```
PersonDAO

This class will help in retrieving and storing person’s records in database. The code is given below:

```java
package vu;
import java.util.*;
import java.sql.*;

public class PersonDAO {
    private Connection con;
    // default constructor

    public PersonDAO() throws ClassNotFoundException, SQLException { establishConnection(); }

    // method used to establish connection with db

    private void establishConnection() throws ClassNotFoundException, SQLException {

        // establishing connection
        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

        String conUrl = "jdbc:odbc:PersonDSN";
        con = DriverManager.getConnection(conUrl);
    }
}
```
// used to search the person records against name and returns
// the ArrayList that contains only those PersonInfo objects
// which matches the search criteria i.e. name

public ArrayList retrievePersonList(String Name) throws SQLException {
    ArrayList personList = new ArrayList();

    // preparing query
    String sql = "SELECT * FROM Person WHERE name = ?";

    PreparedStatement pStmt = con.prepareStatement(sql);
    pStmt.setString(1, pName);

    // executing query
    ResultSet rs = pStmt.executeQuery();

    String name;
    String add;
    int pNo;

    while (rs.next()) {
        name = rs.getString("name");
        add = rs.getString("address");
        pNo = rs.getInt("phoneNumber");

        // creating a CourseOutlineBean object
        PersonInfo personBean = new PersonInfo();
        personBean.setName(name);
        personBean.setAddress(add);
        personBean.setPhoneNumber(pNo);

        // adding a bean to arraylist
        personList.add(personBean);
    } // end while

    return personList;
}
} // end retrievePersonList

// this method accepts an object of PersonInfo, and stores it into // the database
public void addPerson(PersonInfo person) throws SQLException {

    String sql = "INSERT INTO Person VALUES (?, ?, ?)";
    PreparedStatement pStmt = con.prepareStatement(sql);

    String name = person.getName();
    String addr = person.getAddress();
    int pNo = person.getPhoneNum();

    pStmt.setString(1, name);
    pStmt.setString(2, addr);
    pStmt.setInt(3, pNo);

    pStmt.executeUpdate();

} // end addPerson

// overriding finalize method to release acquired resources
public void finalize() {
    try {

        if(con != null) {
            con.close();
        }
        } catch(SQLException sqlex) {
            System.out.println(sqlex);
        }
    } // end finalize

} // end PersonDAO class
Now let's take a look at the code for JSP pages

addperson.jsp

This JSP page gets person record's information from the user. It contains three Input Fields for name, address and phone number as shown in the diagram. This page sends this information to saveperson.jsp for further processing,

![Address Book](image)

The code that is used to generate the above page is given below:

```jsp
<%@page errorPage="addbookerror.jsp" %>
<html> <body>  
<center>  
<h2>Address Book</h2>  
<h3>Add New Person</h3>  

<form name="register" action="saveperson.jsp" />
```

```jsp
<%-- Although there are no chances of exception to arise on this page, for consistency, error page is defined on top of all JSPs--%>
```

```jsp
<%-- Form that contains Text input fields and sending it to saveperson.jsp --%>
```

```jsp
</form>
</center>  
</body>  
</html>
```
<TABLE BORDER="1">
  <TR>
    <TD> <h4> Name </h4> </TD>
    <TD> <input type="text" name="name" /> </TD>
  </TR>
  <TR>
    <TD> <h4> Address </h4> </TD>
    <TD> <input type="text" name="address" /> </TD>
  </TR>
  <TR>
    <TD> <h4> Phone Number </h4> </TD>
    <TD> <input type="text" name="phoneNum" /> </TD>
  </TR>
  <TR>
    <TD COLSPAN="2" ALIGN="CENTER">
      <input type="submit" value="save" />
      <input type="reset" value="clear" />
    </TD>
  </TR>
</TABLE>

<h4>
  <%- -- A link to searchperson.jsp --%>
  <a href="searchperson.jsp"> Search Person </a>
</h4>
saveperson.jsp

This JSP page gets data from the addperson.jsp, makes an object of PersonInfo and saves it to the database using PersonDAO class. Apart from these, it also displays an informative message to the user if new person record is saved successfully into the database and two hyperlinks to navigate on to the desired pages as shown in the following diagram:

![New Person Record is saved successfully!](image)

The code of this page is given below:

```jsp
<%@ page errorPage="addbookerror.jsp" %>
<%@ page import="java.sql.*" %>
<html>
<body>

<%-- defining error page --%>
<%@ page errorPage="addbookerror.jsp" %>
<%@ page import="java.sql.*" %>
<html>
<body>

<%-- creating PersonDAO object and storing in page scope --%>
<jsp:useBean id="pDAO" class="vu.PersonDAO" scope="page" />

<%-- creating PersonBean object and storing in page scope --%>
<jsp:useBean id="personBean" class="vu.PersonInfo" scope = "page" />

<%-- setting all properties of personBean object with input parameters using *
   --%>
<jsp:setProperty name="personBean" property="*" />

<%-- to save Person record into the database, calling addperson
   --%>
```
method of PersonDAO

```html
<%>
pDAO.addPerson(personBean);
<%>

<center>
<h3>New Person Record is saved successfully!</h3>
</center>
<h4><a href="addperson.jsp">Add Person</a></h4>
<h4><a href="searchperson.jsp">Search Person</a></h4>
</center>
</body>
</html>
```

searchperson.jsp

It gets search criteria from the user (i.e. name) and sends it to showperson.jsp to display the search results. The outlook of the page is given below:

![Address Book](image.png)
The code used to generate the above page given is:

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

<html>
<body>
    <center>
        <h2>Address Book</h2>
        <h3>Search Person</h3>
        
        <form name="search" action="showperson.jsp">
            <input type="text" name="name" />
            
            <input type="submit" value="search" />
            <input type="reset" value="clear" />
        </form>
        
        <a href="addperson.jsp">Add Person</a>
    </center>
</body>
</html>
```
showperson.jsp

showperson.jsp receives search criteria (i.e. name) from the searchperson.jsp, that is entered by the user to find the matching record. This page retrieves the complete list of matching records from the database using PersonDAO, and shows them to the user.

This following figure gives you the sight, when person named “saad” is searched.

![Address Book](image)

Below, the code of showperson.jsp is given:

```html
<%@page errorPage="addbookerror.jsp" %>
<%@page import="java.util.*" %>
<%@page import="vu.*" %>

<html>
  <body>
    <center>
      <h2>Address Book</h2>
      <h3>Following results meet your search criteria</h3>

      <table border="1">
        <tr>
          <th>Name</th>
          <th>Address</th>
          <th>PhoneNum</th>
        </tr>
        <tr>
          <td>saad</td>
          <td>gulberg</td>
          <td>9200408</td>
        </tr>
      </table>

      <a>Add Person</a>
      <a>Search Person</a>
    </center>
  </body>
</html>
```
<jsp:useBean id="pDAO" class="vu.PersonDAO" scope="page" />

<% 

// getting search criteria sent by searchperson.jsp

String pName = request.getParameter("name");

// retrieving matching records from the Database using // retrievePersonList() method of PersonDAO

ArrayList personList = personDAO.retrievePersonList(pName);

PersonInfo person = null;

// Showing all matching records by iterating over ArrayList

for(int i=0; i<personList.size(); i++) {
    person = (PersonInfo)personList.get(i);

    <TD> <TD> <%= person.getName() %></TD>
    <TD> <%= person.getAddress() %></TD>
    <TD> <%= person.getPhoneNum() %></TD>

  </TR>

} // end for

</TABLE>

<a href="addperson.jsp" >Add Person</a>  
<a href="searchperson.jsp" >Search Person</a>  
</center>

</body>

</html>
addbookerror.jsp

This JSP error page is called implicitly by all other JSP pages whenever any uncaught / unhandled exception occurs. It also finds out the type of the exception that is generated, and shows an appropriate message to the user:

```html
<%@-- indicating that this is an error page --%>
<%@page isErrorPage="true" %>
<%@-- importing class --%>
<%@page import = "java.sql.SQLException" %>

<html>
<head> <title>Error</title> </head>

<body>
<h2>Error Page</h2>

<h3><%-- scriptlet to determine exception type --%>

<% if (exception instanceof SQLException) {
    %>
    An SQL Exception
<% }
} else if (exception instanceof ClassNotFoundException){
    %>
    A Class Not Found Exception
<% }
else {
    %>
    A Exception
<% }
// end if-else
%>

occured while interacting with the database
```
The Error Message was
<%= exception.getMessage() %>

Please Try Again Later!

<%=-- hyperlinks to return back to addperson.jsp or searchperson.jsp

<a href="controller.jsp?action=addperson" > Add Person </a>

Search Person </a>

Model View Controller (MVC)

Now, more than ever, enterprise applications need to support multiple types of users with multiple types of interfaces. For example, an online store may require an HTML front for Web customers, a WML front for wireless customers, a Java™ (JFC) / Swing interface for administrators, and an XML-based Web service for suppliers.
Also, several problems can arise when applications contain a mixture of data access code, business logic code, and presentation code. Such applications are difficult to maintain, because interdependencies between all of the components cause strong ripple effects whenever a change is made anywhere. High coupling makes classes difficult or impossible to reuse because they depend on so many other classes. Adding new data views often requires re-implementing or cutting and pasting business logic code, which then requires maintenance in multiple places. Data access code suffers from the same problem, being cut and pasted among business logic methods.

The Model-View-Controller architecture solves these problems by decoupling data access, business logic, and data presentation and user interaction. Such separation allows multiple views to share the same enterprise data model, which makes supporting multiple clients easier to implement, test, and maintain.

Participants and Responsibilities

The individual’s responsibility of three participants (model, view & controller) is given below:

. Model

The model represents the state of the component (i.e. its data and the methods required to manipulate it) independent of how the component is viewed or rendered.

. View

The view renders the contents of a model and specifies how that data should be presented. There can be multiple views for the same model within single applications or model may have different views in different applications or operating systems.

. Controller

The controller translates interactions with the view into actions to be performed by the model. In a web application, they appear as GET and POST HTTP requests. The actions performed by the model include activating business processes or changing the state of the model. Based on the user interactions and the outcome of the model actions, the controller responds by selecting an appropriate view.

Evolution of MVC Architecture

In the beginning, we used no MVC. Then we had MVC Model 1 and MVC Model 2 architectures. And people came up with so called web application frameworks such as Apache Struts based on Model 2 architecture. And finally we have a standard web based application framework i.e. JavaServer Faces (JSF).

In this handout, we'll only talk about MVC Model 1.
MVC Model 1

A Model 1 architecture consists of a Web browser directly accessing Web-tier JSP pages. The JSP pages access JavaBeans that represent the application model. And the next view
to display (JSP page, servlet, HTML page, and so on) is determined either by hyperlinks selected in the source document or by request parameters.

In Model 1 architecture, view selection is decentralized, because the current page being displayed determines the next page to display. In addition, each JSP page or servlet processes its own inputs (parameters from GET or POST). And this is hard to maintain, for example, if you have to change the view selection, then several JSP pages need to be changed. In some Model 1 architectures, choosing the next page to display occurs in scriptlet code, but this usage is considered poor form.

In MVC Model 1 architecture, the JSP page alone is responsible for processing the incoming request and replying back to the client. There is still separation of presentation from content, because all data access is performed using JavaBeans.

Although the Model 1 architecture should be perfectly suitable for simple applications, it may not be desirable for complex implementations. Random usage of this architecture usually leads to a significant amount of scriptlets or Java code embedded within the JSP page, especially if there is a significant amount of request processing to be performed. While this may not seem to be much of a problem for Java developers, it is certainly an issue if your JSP pages are created and maintained by designers which are only aware of HTML and some scripting language.

Note: Probably some of you must be thinking about the case study discussed earlier in this handout. Indeed, it is based on MVC Model 1 architecture.

References:

- Java A Lab Course by Umair Javed
- Java BluePrints - J2EE Patterns
- Exploring the MVC Design Pattern
  http://www.javaworld.com/javaworld/jw-12-1999/jw-12-ssj-jspmvc.html
MVC Model 2 Architecture

We have studied page-centric approach and page-with-bean approach until now. You must be wondering when we had covered these. Probably these buzz words are new one for you but we already covered these topics. Let’s review these once again.

Page-Centric Approach

A web application that is collection of JSPs. Generally this approach is followed to get started with developing web applications. This approach is represented in the following diagram:

![Page-Centric Approach Diagram]

The page-centric approach has lot of draw backs such as the code becomes a mixture of presentation, business and data access logic. The maintenance and up-gradation of the application becomes a nightmare. Scaling of such kind of application is also difficult and lots of code is also get duplicated.

Page-with-Bean Approach (MVC Modell)

This approach is different from page-centric approach in a way that all the business logic goes into JavaBeans. Therefore, the web application is a collection of JSPs and JavaBeans. But still this approach is insufficient to separate different kind of logics. We have made an address book example in the last handout using this approach.

![Page-with-Bean Approach Diagram]

MVC Model 2 Architecture

This architecture introduces a controller. This controller can be implemented using JSP or servlet. Introducing a controller gives the following advantages:

- It centralizes the logic for dispatching requests to the next view based on:
  - The Request URL.
- Input Parameters
- Application state

- It gives the single point of control to perform security checks and to record logging information
- It also encapsulates the incoming data into a form that is usable by the back-end MVC model. We’ll discuss it with the help of an example.

The following figure will help you to understand the architecture and functioning of the application that is built using MVC Model 2 architecture.

The client (browser) sends all the requests to the controller. Servlet/JSP acts as the Controller and is in charge of the request processing and creation of any beans or objects (Models) used by the JSP.

JSP is working as View and there is not much processing logic within the JSP page itself, it is simply responsible for retrieving objects and/or beans, created by the Servlet, extracting dynamic content from them and put them into the static templates.

**Case Study: Address Book using MVC Model 2**

The address book example that is built using page-with-bean approach will be modified to incorporate controller. We’ll show you how to implement controller using JSP as well as with servlet. Let’s first incorporate controller using JSP.
Introducing a JSP as Controller

Add another JSP (controller.jsp) that

- Acts as a controller
- Receives requests from addperson.jsp & searchperson.jsp
- Identifies the page which initiates the request
- Uses JavaBeans to save or search persons to/from database
- Forwards or redirects the request to appropriate (saveperson.jsp or showperson.jsp) page.

The program flow of this example is shown in the following diagram:

As you can see in the diagram that all the requests are submitted to controller which uses the JavaBeans and forwards/redirects the user to another view (JSP)? If any exception arises on controller or JSPs, the control would automatically be transferred to addbookerror.jsp to display an appropriate message.

How controller differentiates between requests?

Most likely, you must be thinking about it. The simplest solution lies in using the consistent name (e.g. action) of the submit button across all the pages but with different and unique values. The same rule applies to hyperlinks that send the action parameter along with value by using query string technique.

This eases the controller's job to identify which page is actually generated the request and what to do next. The controller simply retrieves the value of action parameter using request.getParameter() method. Now, if-else structure can be used to compare the possible values of action to act upon the requested task.

Now, let’s first see the code of JavaBean that is used in this example.
PersonInfo

This JavaBean is used to represent one person record. The code is given below:

```java
package vu;
import java.io.*;

public class PersonInfo implements Serializable {
    private String name;
    private String address;
    private int phoneNum;

    // no argument constructor
    public PersonInfo() {
        name = "";
        address = "";
        phoneNum = 0;
    }

    // setters
    public void setName(String n) {
        name = n;
    }

    public void setAddress(String a) {
        address = a;
    }

    public void setPhoneNum(int pNo) {
        phoneNum = pNo;
    }

    // getters
    public String getName() {
        return name;
    }

    public String getAddress() {
        return address;
    }

    public int getPhoneNum() {
        return phoneNum;
    }
}
```

// end class PersonInfo
PersonDAO

This class will help in retrieving and storing person’s records in database. The code is given below:

```java
package vu;

import java.util.*; import java.sql.*;

public class PersonDAO {
    private Connection con;

    // default constructor

    public PersonDAO() throws ClassNotFoundException, SQLException {
        establishConnection();
    }

    // method used to establish connection with db

    private void establishConnection() throws ClassNotFoundException, SQLException {
        // establishing connection
        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
        String conUrl = "jdbc:odbc:PersonDSN";
        con = DriverManager.getConnection(conUrl);
    }

    // used to search the person records against name and returns the ArrayList that contains only
    // those PersonInfo objects which matches the search criteria i.e. name

    public ArrayList retrievePersonList(String pName) throws SQLException {
        ArrayList personList = new ArrayList();

        // preparing query
        String sql = " SELECT * FROM Person WHERE name = ?";

        PreparedStatement pStmt = con.prepareStatement(sql);
        pStmt.setString(1, pName);

        // executing query
        ResultSet rs = pStmt.executeQuery();
    }
}
```
String name;
String add;
int pNo;

while ( rs.next() ) {
    name = rs.getString("name");
    add = rs.getString("address");
    pNo = rs.getInt("phoneNumber");

    // creating a CourseOutlineBean object
    PersonInfo personBean = new PersonInfo();
    personBean.setName(name);
    personBean setAddress(add);
    personBean.setPhoneNum(pNo);

    // adding a bean to arraylist
    personList.add(personBean);
} // end while

return personList;
} // end retrievePersonList

// this method accepts an object of PersonInfo, and stores it into
// the database

public void addPerson(PersonInfo person) throws SQLException {

    String sql = " INSERT INTO Person VALUES (?, ?, ?) " ;
    PreparedStatement pStmt = con.prepareStatement(sql);
    String name = person.getName();
    String add = person.getAddress();
    int pNo = person.getPhoneNum();

    pStmt.setString(1, name);
    pStmt.setString(2, add);
    pStmt.setInt(3, pNo);

    pStmt.executeUpdate();
} // end addPerson  // overriding finalize method to release acquired resources
public void finalize() {
    try{
        con.close();
    }

    }catch(SQLException sqlx){
        System.out.println(sqlx);
    }
} // end finalize

} // end PersonDAO class

adddperson.jsp

This page is used for entering a new person record into the database. Note that a hyperlink is also given at the bottom of the page that takes the user to searchperson.jsp.

Note: Since we are following MVC model 2 architecture, so all the hyperlinks will also sends the request to controller first which redirects the user to requested page.

The code of above page is given below:
<%-- Although there are no chances of exception to arise on this page, for consistency, error page is defined on top of all JSPs
--%>

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

<html>

<body>

<center>

<h2>Addreses Book <h2>

<h3>Add New Person<h3>

<%-- As mentioned in MVC2, all the requests are submitted to controller, that's why action's contains the value of “controller.jsp”
--%>

<form name="register" action="controller.jsp" />

<TABLE BORDER="1">

<TR> <TD> <h4>Name</h4>   </TD> <TD> <input type="text" name="name" /> </TD> </TR>

<TR> <TD> <h4>Address</h4> </TD> <TD> <input type="text" name="address" /> </TD> </TR>

<TR> <TD> <h4>Phone Number</h4> </TD> <TD> <input type="text" name="phoneNum" /> </TD> </TR>

<TR> <TD COLSPAN="2" ALIGN="CENTER"> <input type="submit" name="action" value="save" /> </TD> </TR>

<input type="reset" value="clear" />
</TD>
</TR>

</TABLE>

</center>

</body>

</html>
<h4>
<%-
   The hyperlink will also sends the request to controller
   Note the action parameter with its value are also part of 'hyperlink using
   the query string technique.
-%>
</h4>

<a href="controller.jsp?action=searchperson" >Search Person</a>
searchperson.jsp

This JSP is used to search the person record against name given in the text field. A hyperlink is also given at the bottom of addperson.jsp.

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

<html>
<body>

<center>

<h2>Address Book</h2>
<h3>Search Person</h3>

<form name="search" action="controller.jsp" />

<TABLE BORDER="1" >

 <TR> <TD> <h4> Name </h4> </TD> <TD> <input type="text" name="name" / > </TD>

</TR>

</TABLE>

</form>

Add Person

```

The code that is used to generate that above page is given below:
<%- The name of the button is still “action” but with different value “search”.

-%>

<input type="submit" name="action" value="search" />

<input type="reset" value="clear" />

</TD>
</TR>
</TABLE>
</form>

<h4>

<%- The action parameter with different value “addperson” are part of hyperlink here as well.

-%>

<a href="controller.jsp?action=addperson" > Add Person </a>
</h4>

</center>

</body>

</html>
controller.jsp

As mentioned earlier that controller.jsp identifies the page which initiates the request and use JavaBeans to save/search persons to/from database. Also its job list includes redirecting the user to appropriate page.

Since this JSP is doing only processing therefore no view available. Let’s check it out its code:

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

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

<html>
<body>

<%-- declaring PersonDAO object--%>
<jsp:useBean id="pDAO" class="vu.PersonDAO" scope="page" />

<%@-- scriptlet to identify JSP for redirection purpose if request comes from hyperlinks --%>

// retrieving action parameter value
// Remember that “action” is the name of buttons as well
// it is used in hyperlinks in making of query string

String action = request.getParameter("action");

// if "Add Person" hyperlink is clicked
if (action.equals("addperson")){
    response.sendRedirect("addperson.jsp");
}

// if "Search Person" hyperlink is clicked
} else if (action.equals("searchperson")){
    response.sendRedirect("searchperson.jsp");

// if "save" button is clicked of addperson.jsp
} else if (action.equals("save")) {

%>
```
// declaring PersonInfo object

<jsp:useBean id="personBean" class="vu.PersonInfo" scope="page"/>

<%--
    setting all properties of personBean object with input
    parameters using *
--%>

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

<%-- to insert record into database--%>
<% pDAO.addPerson(personBean);

    // redirecting user to saveperson.jsp
    response.sendRedirect("saveperson.jsp"); %>

<%-- if "search" button is clicked on searchperson.jsp --%>
<% } else if (action.equals("search") ) { %>

    String pName = request.getParameter("name");  ArrayList personList = pDAO.retrievePersonList(pName);
    // storing personList(contains PersonInfo objects) into // request hashmap
    request.setAttribute("list", personList);

    <%-- forwarding request to showperson.jsp to retrieved stored arraylist ("list") --%>

    <jsp:forward page="showperson.jsp" />

<% } // end if page == search %>

</body>
</html>
saveperson.jsp

This page displays a successful message indicating that person record is saved. It also gives the options to the user to move on to addperson.jsp or searchperson.jsp through hyperlinks. Note that these hyperlinks also first take the user to controller.jsp then on to requested page.

The code of saveperson.jsp is given below:

```html
<%@ page errorPage = "addbookerror.jsp" %>
<html>
  <head></head>
  <body>
    <center>
      <h3>New Person Record is saved successfully!</h3>
      <h4>
        <a href="controller.jsp?action=addperson">Add Person</a>
      </h4>
      <h4>
        <a href="controller.jsp?action=searchperson">Search Person</a>
      </h4>
    </center>
  </body>
</html>
```
This following figure gives you the view when name "saad" is searched.

Below, the code of showperson.jsp is given:

```jsp
<%@-- defining error page --%>
<%@page errorPage="addbookerror.jsp" %>

<%@-- importing required packages --%>
<%@page import="java.util.*" %>
<%@page import="vu.*" %>

<html> <body>

    <center>
        <h2> Address Book </h2>
        <h3> Following results meet your search criteria </h3>

        <TABLE BORDER="1">
            <TR>
                <TH> Name </TH>
                <TH> Address </TH>
                <TH> PhoneNum </TH>
            </TR>
            <TR>
                <TD> saad </TD>
                <TD> gilberg </TD>
                <TD> 9200408 </TD>
            </TR>
        </TABLE>

        <a href="addperson.jsp"> Add Person </a>
        <a href="search.jsp"> Search Person </a>

    </center>

</body>
</html>

// retrieving ArrayList stored on controller.jsp to display PersonInfo objects
ArrayList personList = (ArrayList)request.getAttribute("list"); PersonInfo person = null;

for(int i=0; i<personList.size(); i++) {
    person = (PersonInfo)personList.get(i);
    <\%-- displaying PersonInfo details--\%>
    <TR><TD><% person.getName()%></TD> <TD><% person.getAddress()%></TD> <TD><% person.getPhoneNum()%></TD></TR>

    <\%}
    // end for

    \%

</\TABLE>

<h4>

<a href="controller.jsp?action=addperson">Add Person</a> <a href="controller.jsp?action=searchperson">Search Person</a>

</h4>

</center>

</body>

</html>
addbookerror.jsp

User will view this page only when any sort of exception is generated. The code of this page is given below:

<%@ page isErrorPage="true" %>

<%@ page import = "java.sql.SQLException" %>

<html>
   <head>  
      <title>Error</title>  
   </head>

   <body>
      
      <h2>
         Error Page
      </h2>

      <h3>
         <%-- scriptlet to determine exception type --%>

         <%
            if (exception instanceof SQLException) {
               
               An SQL Exception
            
            } else if (exception instanceof ClassNotFoundException) {
               
               A Class Not Found Exception
            
            } else {
               
               A Exception
            
            } // end if-else
         %>

         <%-- end scriptlet to determine exception type --%>
   </body>
</html>
occurred while interacting with the database

</h3>

<h3>The Error Message was
<%= exception.getMessage() %>

</h3>

<h3>Please Try Again Later! </h3>

<%-- hyperlinks to return back to adperson.jsp or searchperson.jsp --%>

<h3>

<a href="controller.jsp?action=addperson" >Add Person </a>

<a href="controller.jsp?action=searchperson" >Search Person </a>

</h3>

</body>
</html>

JSP is the Right Choice as a Controller?

Since JSP that is performing the job of controller is doing only processing and there is no view available of it. It includes the logic of selecting JSP and to retrieve/store records from/to dataset using JavaBeans.

But remember the reason for introducing JSPs? JavaServer Pages are built for presentation (view) only so JSP is really not a good place for such kind of logic. Concluding, what’s the option we have? The answer is, use Servlets as controller.

Introducing a Servlet as Controller

Remove the controller.jsp from the previous example code and add ControllerServlet.java (a servlet) into this example. This ControllerServlet.java performs the same job that was previously performed by controller.jsp.

Besides adding ControllerServlet.java, you have to modify all the addresses which are previously pointing to controller.jsp. For example the value of action attribute of form tag & the address of hyperlink in all
concerned pages.

If controller is defined in web.xml as an alias of ControllerServlet.java, consider the following fragment of code which shows the value of action attribute of form tag before and after introducing change.

When controller.jsp is acting as a controller

```
<form name="register" action="controller.jsp" />
```

When ControllerServlet.java is acting as a controller then value of action attribute becomes:

```
<form name="register" action="controller" />
```

Similarly, the following comparison shows the code of hyperlinks used in the previous example before and after making changes

When controller.jsp is acting as a controller

```
<a href="controller.jsp?action=searchperson">
Search Person
</a>
```

When ControllerServlet.java is acting as a controller

```
<a href="controller?action=searchperson">
Search Person
</a>
```

**Passing Exceptions to an Error JSP from a Servlet**

Servlet can use existing error pages (like addbookerror.jsp) to pass on the exceptions. Set the request attribute to javax.servlet.jsp.JspException with the exception object you want to pass. After that forwards the request to error page.

For example, the following code snippet is taken from ControllerServlet.java to demonstrate how to pass SQLException to addbookerror.jsp

```
............... 
............... 
} catch (SQLException sqlex) {
    // setting SQLException instance
    request.setAttribute("javax.servlet.jsp.JspException", sqlex);
    RequestDispatcher rd =
    request.getRequestDispatcher("addbookerror.jsp");
    rd.forward(request, response);

} // end catch
```
ControllerServlet.java

The following code is of servlet that is acting as a controller

package controller;
import vu.*;
import java.io.*; import java.net.*;
import java.sql.*; import java.util.*;
import javax.servlet.*; import javax.servlet.http.*;

public class ControllerServlet extends HttpServlet {

    // This method only calls processRequest()
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        processRequest(request, response);
    }

    // This method only calls processRequest()
    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        processRequest(request, response);
    }

    protected void processRequest(HttpServletRequest request, HttpServletResponse response) {

        // retrieving value of action parameter
        String userAction = request.getParameter("action");

        // if request comes to move to addperson.jsp from hyperlink
        if (userAction.equals("addperson")) {
            response.sendRedirect("addperson.jsp");
        }

        // if request comes to move to searchperson.jsp from hyperlink
        } else if (userAction.equals("searchperson")) {
            response.sendRedirect("searchperson.jsp");
        }
    }
}
// if “save” button clicked on addperson.jsp to add new record

} if (userAction.equals("save")) {
    // this method defined below
    addPerson(request, response);

    // if “search” button clicked on searchperson.jsp for search

} else if (userAction.equals("search")) {
    // this method defined below
    searchPerson(request, response);
}

} // end processRequest()

// if request comes to add/save person

private void addPerson(HttpServletRequest request, ServletResponse response)
    throws ServletException, IOException {
    try {
        // creating PersonDAO object
        PersonDAO pDAO = new PersonDAO();

        // creating PersonInfo object
        PersonInfo person = new PersonInfo();

        // setting properties of Person object
        // setting name property
        String pName = request.getParameter("name");
        person.setName(pName);

        // setting address property
        String add = request.getParameter("address");
        person.setAddress(add);

        // setting phoneNumber property
        String pNo = request.getParameter("phoneNum");
        int phoneNum = Integer.parseInt(pNo);
        person.setPhoneNumber(phoneNum);

        // calling PersonDAO method to save data into database
        pDAO.addPerson(person);

        // redirecting page to saveperson.jsp
        response.sendRedirect("saveperson.jsp");
    }
catch (SQLException sqlex) {

    // setting SQLException instance
    request.setAttribute("javax.servlet.jsp.JspException", sqlex);
    RequestDispatcher rd = request.getRequestDispatcher("addbookerror.jsp");

    rd.forward(request, response);  }

catch (ClassNotFoundException cnfe) {

    // setting ClassNotFoundException instance
    request.setAttribute("javax.servlet.jsp.JspException", cnfe);
    RequestDispatcher rd = request.getRequestDispatcher("addbookerror.jsp"); rd.forward(request, response);

}  }  // end addperson()

// if request comes to search person record from database

private void searchPerson(HttpServletRequest request,

    HttpServletResponse response)  throws ServletException, IOException {

    try {

        // creating PersonDAO object
        PersonDAO pDAO = new PersonDAO();
        String pName = request.getParameter("name");

        // calling DAO method to retrieve personlist from database  // against name
        ArrayList personList = pDAO.retrievePersonList(pName);
        request.setAttribute("list", personList);

}
// forwarding request to showperson, so it can render personlist
RequestDispatcher rd = request.getRequestDispatcher("showperson.jsp");

rd.forward(request, response);
}
catch (SQLException sqlex) {
    // setting SQLException instance
    request.setAttribute("javax.servlet.jsp.JspException", sqlex);
    RequestDispatcher rd = request.getRequestDispatcher("addbookerror.jsp");
    rd.forward(request, response);
}
catch (ClassNotFoundException cnfe) {
    // setting ClassNotFoundException instance
    request.setAttribute("javax.servlet.jsp.JspException", cnfe);
    RequestDispatcher rd = request.getRequestDispatcher("addbookerror.jsp");
    rd.forward(request, response);
}
} // end searchPerson()
} // end ControllerServlet

web.xml

As you already familiar, for accessing a servlet, you need to define a URL pattern in web.xml. This is shown below:

<?xml version="1.0" encoding="UTF-8"?> <web-app>
    <servlet>
        <servlet-name>ControllerServlet</servlet-name>
        <servlet-class>controller.ControllerServlet</servlet-class>
    </servlet>
<servlet-mapping>
    <servlet-name>ControllerServlet</servlet-name>
    <url-pattern>/controller</url-pattern>
</servlet-mapping>

</web-app>

References:

. Java A Lab Course by Umair Javed.
. Java E-commerce course at Stanford
Lesson 41

Layers and Tiers

How do you structure an application to support such operational requirements as maintainability, reusability, scalability and robustness? The answer lies in using Layers and Tiers. What different technologies Java provides to support layered or tiered architectures. The answer to these questions will remain our focus in this handout. A small case study will also be used to comprehend the concept of layers.

Layers vs. Tiers

Layers are merely logical grouping of the software components that make up the application or service, whereas Tiers refer to the physical residence of those layers.

In general,

- **Layers** – represents the logical view of application
- **Tiers** – represents physical view of application

However, both terms are used intractably very often. You must be confused what does logical & physical view mean? Let’s elaborate layers and tiers further in detail to differentiate between them.

Layers

The partitioning of a system into layers such that each layer performs a specific type of functionality and communicates with the layer that adjoin it.

The separation of concerns minimizes the impact of adding services/features to an application. The application developed in layers also enables tiered distribution (discussed later). Furthermore easier maintenance, reuse of code, high cohesion & loose coupling sort of additional benefits are also enjoyed by the use of tiered architecture.

To begin with, layered architecture based on three layers. These are

- Presentation Layer
- Business Layer
- Data Layer

Note:

However, there is no upper limit of number of layers an application can have. Each layer can also be further break down into several layers depending upon the requirement and size of the application.
The figure given below shows a simplified view of an application and its layers.

As you can see in the figure, users can only interact with the presentation layer. The presentation layer passes the user request to the business layer, which further passes the request to the data layer. The data layer communicates with the data sources (like Database etc.) or other external services in order to accomplish the user request.

Let's discuss each layer's responsibility in detail:

**Presentation Layer**

It provides a user interface to the client/user to interact with the application. This is the only part of the application visible to client.

Its job list includes collecting user's input, validating user's input (on client side using JavaScript like technologies OR on server side), presenting the results of the request made by the user and controlling the screen flow (which page/view will be visible to the user).

**Business Layer**

Also called application layer, it is only concerned with the application specific functionality. It is used to implement business rules and to perform business tasks.

For example, in a banking system, this layer will provide the functionality of banking functions such as opening an account, transferring of balance from one account to another, Calculation of taxes etc.
Data Layer

It is concerned with the management of the data & data sources of the system. Data sources can be database, XML, web services, flat file etc. Encapsulates data retrieval & storage logic For example, the address book application needs to retrieve all person records from a database to display them to the user.

Tiers

As mentioned, layers help in building a tiered architecture. Like layers, there is no restriction on using number of tiers. An application can be based on Single-tier, Two-tier, Three-tier or N-Tier (application which have more than three tiers). The choice of using a tiered architecture is contingent to the business requirements and the size of the application etc.

Tiers are physically separated from each other. Layers are spread across tiers to build up an application. Two or more layers can reside on one tier. The following figure presents a three-tier architectural view of an application.

![Three-tier architecture diagram]

The client tier represents the client machine where actually web browser is running and usually displays HTML. You can think of a Presentation as of two parts; one is on client side, for example, HTML. There is also a presentation layer that is used to generate the client presentation often called server presentation. We’ll discuss about it later.

The server machine can consist on a single server machine or more. Therefore, it is possible web server is running on one server machine while application server on another. Web server is used to execute web pages like JSPs whereas application server is used to run special business objects like Enterprise JavaBeans (discussed later). The web layer and applications server can be on two separate machines or they can be on same tier as shown in the diagram.

The client tier represents the client machine often called Enterprise information tier.
Layers Support in Java

The secret of wide spread use of Java lies in providing specific technology for each layer. This not only eases the development by freeing the programmer for caring operational features but only reduces the production time of the software.

In the following figure, Presentation is bifurcated into two layers. These are Client Presentation layer and Server Presentation Layer. What client sees in a browser forms client presentation layer while server presentation layer includes the Java technology components?

(JSP and Servlets etc.) that are used to generate the client presentation.

Layer Java/J2EE Technology

<table>
<thead>
<tr>
<th>Client Presentation</th>
<th>HTML/Applets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Server Presentation</td>
<td>JSP / Servlets Frameworks (Struts, JSF etc)</td>
</tr>
<tr>
<td>Business</td>
<td>JavaBeans / EJB</td>
</tr>
<tr>
<td>Data</td>
<td>DAO / Connectors</td>
</tr>
</tbody>
</table>

On business layer, JavaBeans (also referred as Plain Old Java Objects (POJO) ) can be used. While moving towards a bigger architecture, the J2EE provides the special class that fits in business layer i.e. Enterprise JavaBean (EJB).

EJBs are special java classes that are used to encapsulate business logic. They provide additional benefits in building up an application such as scalability, robustness, scalability etc.

On data layer, Data Access Objects (DAO) can be used. Similarly you can use connectors. There are other different specialized components provided in java that ease the development of data layer.
J2EE Multi-Tiered Applications

In a typical J2EE Multi-Tiered application, a client can either be a swing based application or a web based. As you can see in the following figure, clients can access the web server from behind the firewall as well.

Suppose, our client is HTML based. Client does some processing on HTML and transports it to web server. JSP and Servlets are possible technologies that can be used in a web server. However, there are some Frameworks such as JSF etc that can be used in a web server. The classes which form the presentation layer reside on web server and of course controllers are also used over here.

![J2EE Multi-Tiered Applications Diagram]

If web server, wants to perform some business process, it usually gets help from some business layer components. The business layer component can be a simple JavaBean (POJO) but in a typical J2EE architecture, EJBS are used. Enterprise JavaBeans interacts with the database or information system to store and retrieve data.

EJBS and JSP/Servlets works in two different servers. As you already know, JSP and Servlets runs in a web server where as EJBS requires an application server. But, generally application server contains the web server as well.

Application server including web server generally resides on a single tier (machine), which is often called middle tier. This tier stores and retrieves data from the Enterprise Information Tier (EIS) which is a separate tier. The response sends back to the client by the middle tier can be HTML, XML etc. This response can be seen on the separate tier know as client tier.
Case Study: Matrix Multiplication using Layers

Problem Statement

. Calculate product of two matrices of order 2 * 2
. Result of multiplication should be stored in DB as well as shown to the user.

Format

. Input format
   -input will be in 4,2,6,5 format separated by commas where 4,2 represents entries of the first row

. Display format
   -Displays the matrix as a square

. Storage format for DB
   -Matrix will be stored as a string in the database along with the order of the matrix
   -The following figure shows the table design that will be used to store the results.

Layer by Layer View

A picture’s worth than thousand words. Therefore, before jumping on to code, let’s put a glance over layers that will be used in this small case study. The classes that will be used on each layer and what functionality each class will perform will also be discussed.

First, look on the following picture that will describe the whole story.
The data layer has a class MatrixDAO that is used to save the matrix result into database. As mentioned in the problem statement, that resultant matrix should be saved in the database. So, MatrixDAOis used to accomplish that.

MatrixDAO called by the MatrixMultiplier, a business layer class. The functionality list of MatrixMultiplier includes:

- Converting the user input string (e.g. 2, 3, 4, 1) into a proper object i.e. a matrix
- data structure.
- Helps in calculating product of two matrices

Controller layer’s class ControllerServlet calls the MatrixMultiplier. This layer calls the various business methods (like multiplication of two matrices) of business layer class and got the resultant matrix. Furthermore, ControllerServlet sends the output to the matrixresult.jsp and receives the input from matrixinput.jsp.

The MatrixBean representing matrix data structure, as you can see in the figure is used across several layers. In fact, the object formed by MatrixMultiplier from a user input string is of MatrixBean type. It is used to transfer data from one layer to another.

First, look on the MatrixBean code given below:
MatrixBean

package bo;

import java.io.*;

public class MatrixBean implements Serializable {

    // a 2D array representing matrix
    public int matrix[] [] ;

    // constructor
    public MatrixBean()
    {
        matrix = new int[2][2];

        matrix[0][0] = 0;
        matrix[0][1] = 0;
        matrix[1][0] = 0;
        matrix[1][1] = 0;
    }

    // setter that takes 4 int values and assigns these to array
    public void setMatrix(int w, int x, int y, int z)
    {
        matrix[0][0] = w;
        matrix[0][1] = x;
        matrix[1][0] = y;
        matrix[1][1] = z;
    }

    // getter returning a 2D array
    public int[][] getMatrix()
    {
        return matrix;
    }

    // used to convert 2D array into string
    public String toString()
    {
        return matrix[0][0] + "," + matrix[0][1] + "," + matrix[1][0] + "," + matrix[1][1];
    }

} // end MatrixBean
matrixinput.jsp

This JSP is used to collect the input for two matrices in the form of string such as 2,3,5,8. The data will be submitted to ControllerServlet from this page.

```html
<html>
<body>

<h2>
Enter Two Matrices of order 2 * 2 to compute Product</h2>

<h3>
<form name="matrixInput" action="controller">

First Matrix:
<input type="text" name="firstMatrix" value="E.g. 2,3,4,1" />
<br/>

Second Matrix:
<input type="text" name="secondMatrix" />
<br/>

<input type = "submit" value = "Calculate Product" />

</form>
</h3>

</body>
</html>
```

ControllerServlet

This servlet acting as a controller receives the input from matrixinput.jsp. Furthermore, it will interact with the business layer class MatrixMultiplier to convert the string into a MatrixBean object, and to multiply two matrices.
package controller;

import bl.*;
import bo.*;
import java.io.*;
import java.net.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ControllerServlet extends HttpServlet {

// This method only calls processRequest()
protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
    processRequest(request, response);
}

// This method only calls processRequest()
protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
    processRequest(request, response);
}

protected void processRequest(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {

    // retrieving values from input fields of matrixinput.jsp
    String sMatrix1 = request.getParameter("firstMatrix");
    String sMatrix2 = request.getParameter("secondMatrix");

    // Creating MatrixMultiplier object
    MatrixMultiplier mm = new MatrixMultiplier();

    // Passing Strings to convertToString() method of MatrixMultiplier  // convertToString() is used to convert strings into MatrixBean
MatrixBean fMatrix = mm.convertToObject(sMatrix1);
MatrixBean sMatrix = mm.convertToObject(sMatrix2);

// passing MatrixBean’s objects to multiply() method of MatrixMultiplier and receiving the
// product matrix in the form of MatrixBean
MatrixBean rMatrix = mm.multiply(fMatrix, sMatrix);

// saving results in database
mm.saveResult(rMatrix);

// storing the product of matrices into request, so that it can be retrieved on matrixresult.jsp
req.setAttribute("product", rMatrix);

// forwarding request to matrixresult.jsp
RequestDispatcher rd = req.getRequestDispatcher("matrixresult.jsp");
rd.forward(req, res);

} // end processRequest()
} // end ControllerServlet

MatrixMultiplier
The business layer class that’s primary job is to calculate product of tow matrices given in the form of
MatrixBean. This class also has a method convertToObject that takes a String and returns back a MatrixBean object. MatrixMultiplier will also interact with the data layer class MatrixDAOto store results in
the database.

class MatrixMultiplier {

//constructor

public MatrixMultiplier() {
}

// used to convert a String (like 2,3,4,5) into a MatrixBean object

public MatrixBean convertToObject(String sMatrix) {

}
String tokens[] = sMatrix.split(",");

MatrixBean matrixBO = new MatrixBean();

int w = Integer.parseInt(tokens[0]);
int x = Integer.parseInt(tokens[1]);
int y = Integer.parseInt(tokens[2]);
int z = Integer.parseInt(tokens[3]);

matrixBO.setMatrix(w, x, y, z);
return matrixBO;

} // end convertToobject()

// used to multiply two matrices, receives two MatrixBean objects
// and returns the product in the form of MatrixBean as well

public MatrixBean multiply(MatrixBean fMatrix, MatrixBean sMatrix) {
    MatrixBean resultMatrix = new MatrixBean();

    int matrixA[][] = fMatrix.getMatrix();
    int matrixB[][] = sMatrix.getMatrix();
    int matrixC[][] = resultMatrix.getMatrix();

    // code to multiply two matrices
    for (int i=0; i<2; i++) {
        for (int j=0; j<2; j++) {
            for (int k=0; k<2; k++) {
                matrixC[i][j] += (matrixA[i][k] * matrixB[k][j]);
            }
        }
    }

    // storing the product from 2d array to MatrixBean object by calling setter

resultMatrix.setMatrix( matrixC[0][0], matrixC[0][1], matrixC[1][0], matrixC[1][1] );

return resultMatrix;

} // end MatrixMultiplier

} // end multiply

// save results (MatrixBean containing product of two matrices) into // database using DAO
public void saveResult( MatrixBean resultMatrix )
{

MatrixDAO dao = new MatrixDAO();
    dao.saveMatrix(resultMatrix);

}

} // end MatrixMultiplier


MatrixDAO

As class name depicts, it is used to store product results into database. Let’s look on the code to see how it is accomplished.

package dal;

import java.util.*; import java.sql.*;

import bo.*; public class MatrixDAO{

    private Connection con;

    // constructor public MatrixDAO() throws ClassNotFoundException, SQLException {
        establishConnection();
    }

    // method used to establish connection with db private void establishConnection() throws
        ClassNotFoundException, SQLException {

            // establishing connection
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
        String conUrl = "jdbc:odbc:MatrixDSN";
        con = DriverManager.getConnection(conUrl);
    }
// used to store MatrixBean into database after converting it to // a String
public void saveMatrix(MatrixBean matrix) {
    try {
        String sql = "INSERT INTO Matrix(mOrder, mValues) VALUES (?,?)";
        PreparedStatement pStmt = con.prepareStatement(sql);

        // converting MatrixBean into String by calling toString()
        String sMatrix = matrix.toString();

        // setting order of matrix
        pStmt.setString(1, "2*2");

        // setting matrix values in the form of string
        pStmt.setString(2, sMatrix);

        pStmt.executeUpdate();
    } catch (SQLException sqlex) {
        System.out.println(sqlex);
    }
}

// overriding finalize method to release acquired resources
public void finalize() {
    try {
        if (con != null) {
            con.close();
        }
    } catch (SQLException sex) {
        System.out.println(sex);
    }
}

} // end finalise

} // end MatrixDAO class
matrixresult.jsp

Used to display resultant product of two matrices. The code is given below:

<%@ page import="bo.*"%>

<html>
<body>

<h1>The resultant Matrix is </h1>

<%-- retrieving MatrixBean object from request, that was set on ControllerServlet --%>

<% MatrixBean productMatrix = (MatrixBean)request.getAttribute("product");

// retrieving values in 2d array so that it can be displayed
int matrix[][] = productMatrix.getMatrix(); %>

<%-- displaying MatrixBean’s object values --%>

<TABLE>
  <TR>
    <TD><%= matrix[0][0] %></TD>
    <TD><%= matrix[0][1] %></TD>
  </TR>
  <TR>
    <TD><%= matrix[1][0] %></TD>
    <TD><%= matrix[1][1] %></TD>
  </TR>
</TABLE>
</body>
</html>
web.xml

<?xml version="1.0" encoding="UTF-8"?>

<web-app>
    <servlet>
        <servlet-name>ControllerServlet</servlet-name>
        <servlet-class>controller.ControllerServlet</servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>ControllerServlet</servlet-name>
        <url-pattern>/controller</url-pattern>
    </servlet-mapping>

</web-app>

References:

. Java A Lab Course by Umair Javed.
. Java Passion by Sang Shin
Expression Language

Sun Microsystems introduced the Servlet API, in the later half of 1997, positioning it as a powerful alternative for CGI developers who were looking around for an elegant solution that was more efficient and portable than CGI (Common Gateway Interface) programming. However, it soon became clear that the Servlet API had its own drawbacks, with developers finding the solution difficult to implement, from the perspective of code maintainability and extensibility. It is in some ways, this drawback that prompted the community to explore a solution that would allow embedding Java Code in HTML – Java Server Pages (JSP) emerged as a result of this exploration.

Java as the scripting language in JSP scares many people particularly web page designers which have enough knowledge to work with HTML and some scripting language, faced lot of difficulties in writing some simple lines of java code. Can we simplify this problem to ease the life of web designer? Yes, by using Expression Language (EL).

JavaServer Pages Standard Tag Library (JSTL) 1.0 introduced the concept of the EL but it was constrained to only the JSTL tags. With JSP 2.0 you can use the EL with template text.

**Note:** - JSTL will be discussed in the following Handout.

**Overview**

The Expression Language, not a programming or scripting language, provides a way to simplify expressions in JSP. It is a simple language that is geared towards looking up objects, their properties and performing simple operations on them. It is inspired form both the ECMAScript and the XPath expression language.

**JSP Before and After EL**

To add in motivational factor so that you start learning EL with renewed zeal and zest, a comparison is given below that illustrates how EL affects the JSPs.

The following figure depicts the situation of a JSP before EL. We have to declare a variable before using it, data type must be known in advance and most importantly have to use awkward syntax and many more. All these problems are highlighted in the following figure:
Contrary to the above figure, have a look on the subsequent figure that gives you a hint how useful EL can be?

Person Name: $ \{ \text{p.name} \}$

$\text{...}$

$\text{<c:if test = "\$\{p.address == param.add\}"}>$

**Expression Language Nuggets**

We'll discuss the following important pieces of EL. These are:

- Syntax of EL
- Expressions & identifiers
- Arithmetic, logical & relational operators
- Automatic type conversion
- Access to beans, arrays, lists & maps
- Access to set of implicit objects

**EL Syntax**

The format of writing any EL expression is:

$ \{ \text{validExpression} \}$

The valid expressions can consist on these individuals or combination of these given below:

- Literals
• Operators
• Variables (object references)
• Implicit call to function using property name

**EL Literals**
The list of literals that can be used as an EL expression and their possible values are given in the tabular format below:

<table>
<thead>
<tr>
<th>Literals</th>
<th>Literal Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>true or false</td>
</tr>
<tr>
<td>Integer</td>
<td>Similar to Java e.g. 243, -9642</td>
</tr>
<tr>
<td>Floating Point</td>
<td>Similar to Java e.g. 54.67, 1.83</td>
</tr>
<tr>
<td>String</td>
<td>Any string delimited by single or double quote e.g. “hello”, ‘hello’</td>
</tr>
<tr>
<td>Null</td>
<td>Null</td>
</tr>
</tbody>
</table>

Examples of using EL literals are: . ${false} <%- evaluates to false --%> . ${8*3} <%- evaluates to 24 --%>

**EL Operators**
The lists of operators that can be used in EL expression are given below:

<table>
<thead>
<tr>
<th>Type</th>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>+ - * / (div) % (mod)</td>
</tr>
<tr>
<td>Grouping</td>
<td>()</td>
</tr>
<tr>
<td>Logical</td>
<td>&amp;&amp; (and)</td>
</tr>
<tr>
<td>Relational</td>
<td>== (eq) != (ne) &lt; (lt) &gt; (gt) &lt;= (le) &gt;= (ge)</td>
</tr>
<tr>
<td>Empty</td>
<td>The empty operator is a prefix operation used to determine if a value is null or empty. It returns a Boolean value.</td>
</tr>
<tr>
<td>Conditional</td>
<td>?:</td>
</tr>
</tbody>
</table>
Let us look at some examples that use operators as valid expression:

- `$\{ (6*5) + 5 \} \ <\%--\ evaluate\ to\ 35\ --\%>`
- `$\{ x >= \text{min}\} && (x <= \text{max}) \} $
- `$\{\ \text{empty}\ \text{name}\ \} $

Returns true if name is
- Empty string (""),
- Null etc.

**EL Identifiers**

Identifiers in the expression language represent the names of objects stored in one of the JSP scopes: page, request, session, or application. These types of objects are referred to scoped variables throughout this handout.

EL has 11 reserved identifiers, corresponding to 11 implicit objects. All other identifiers assumed to refer to scoped variables.

**EL implicit Objects**

The Expression Language defines a set of implicit objects given below in tabular format:

<table>
<thead>
<tr>
<th>Category</th>
<th>Implicit Object</th>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>JSP</td>
<td>pageContext</td>
<td>The context for the JSP page, used to access the JSP implicit objects such as request, response, session, out, servletContext etc.</td>
</tr>
<tr>
<td>Scopes</td>
<td>pageScope</td>
<td>A Map associating names &amp; values of page scoped attributes</td>
</tr>
<tr>
<td></td>
<td>requestScope</td>
<td>A Map associating names &amp; values of request scoped attributes</td>
</tr>
<tr>
<td></td>
<td>sessionScope</td>
<td>A Map associating names &amp; values of session scoped attributes</td>
</tr>
<tr>
<td></td>
<td>applicationScope</td>
<td>A Map associating names &amp; values of application scoped attributes</td>
</tr>
<tr>
<td>Request Parameters</td>
<td>param</td>
<td>Maps a request parameter name to a single String parameter value.</td>
</tr>
<tr>
<td></td>
<td>paramValues</td>
<td>Maps a request parameter name to an array of values</td>
</tr>
<tr>
<td>Request Headers</td>
<td>header</td>
<td>Maps a request header name to a single header value.</td>
</tr>
<tr>
<td></td>
<td>headerValues</td>
<td>Maps a request header name to an array of value.</td>
</tr>
<tr>
<td>Cookies</td>
<td>cookie</td>
<td>A Map storing the cookies accompanying the request by name</td>
</tr>
</tbody>
</table>
Examples of using implicit objects are:

- \${\ pageContext.response} - Evaluates to response implicit object of JSP

- \${\ param.name} - This expression is equivalent to calling request.getParameter("name");

- \${\ cookie.name.value} - Returns the value of the first cookie with the given name

- Equivalent to if (cookie.getName().equals("name") { String val = cookie.getValue(); }

**Example Code: Summation of Two Numbers using EL**

This simple example demonstrates you the capabilities of EL. index.jsp is used to collect input for two numbers and their sum is displayed on result.jsp using EL. Let’s first see the code of index.jsp

**index.jsp**

```html
<html>
  <body>
    Enter two numbers to see their sum
    <form action="result.jsp">
      First Number: <input type="text" name="num1"/>
      Second Number: <input type="text" name="num2"/>
      <input type="submit" value="Calculate Sum"/>
    </form>
  </body>
</html>
```
result.jsp

<html>
<body>

<%-- The code to sum two numbers if we used scriptlet

<
String no1 = request.getParameter("num1");
String no2 = request.getParameter("num2");
int num1 = Integer.parseInt(no1);
int num2 = Integer.parseInt(no2);
%>

Result is: <%= num1 + num2 %>

<%-- implicit Object param is used to access request parameters By Using EL summing two numbers

Result is: ${param.num1 + param.num2} %>

</body></html>

EL Identifiers (cont.)

We had started our discussion on EL identifiers. Let’s find out how these identifiers (variables) can be stored/retrieved in/from different scopes.

Storing Scoped Variables

By using java code, either in pure servlet or in a scriptlet of JSP, we can store variables in a particular scope. For example,

. Storing a variable in session scope using Java code

Assume that we have PersonInfo class and we want to store its object in session scope then we can write the following lines of code to accomplish that:

HttpSession ses = request.getSession(true); PersonInfo p = new PersonInfo(); p.setName("ali"); ses.setAttribute("person", p);

. Storing a variable in request scope using Java code

For the following lines of code, assume that request is of HttpServletRequest type. To store PersonInfo object p in request scope, we'll write:
PersonInfo p = new PersonInfo(); p.setName("ali");
request.setAttribute("person", p);

You must be thinking of some another method (with which you are already familiar) to store a variable in a scope, certainly by using JSP action tags, we learned how to store a variable in any particular scope.

. Storing a variable in request scope using JSP action tag
If we want to store p of type PersonInfo in request scope by using JSP action tags, then we'll write:

<jsp:useBean id="p" class="PersonInfo" scope="request"/>

Later, you can change the properties of object p by using action tag as well. For example

<jsp:setProperty name="p" property="name" value="ali"/>

Retrieving Scoped Variables

You are already very much familiar of retrieving any stored scoped variable by using java code and JSP action tags. Here, we'll discuss how EL retrieves scoped variables. As already mentioned, identifiers in the valid expression represent the names of objects stored in one of the JSP scopes: page, request, session and application.

When the expression language encounters an identifier, it searches for a scoped variable with that name first

• in page scope,
• then in request scope,
• then in session scope
• and finally in application scope

Note: - If no such object is located in four scopes, null is returned.

For example, if we've stored PersonInfo object p in session scope by mean of any mechanism discussed previously and have written the following EL expression to access the name property of p

${p.name}

Then EL searches for p first in page scope, then in request scope, then in session scope where it found p. After that it calls p.getName() method. This is also shown in pictorial form below:
EL Accessors

The dot (.) and bracket ([ ] ) operator let you access identifies and their properties. The dot operator typically used for accessing the properties of an object and the bracket operator is generally used to retrieve elements of arrays and collections.

. Dot (.) operator

Assume that JavaBean PersonInfo has name property and its object person is stored in some scope. Then to access the name property of person object, we'll write the following expression using EL:

```
${person.name}
```

The EL accesses the object’s properties using the JavaBeans conventions therefore getName() must be defined in PersonInfo. Moreover, if property being accessed itself an object, the dot operator can be applied recursively. For example

```
$user.address.city
```
Bracket ([ ]) operator

This operator can be applied to arrays & collections implementing List interface e.g. ArrayList etc.

- Index of the element appears inside brackets
- For example, \$\{ personList[2] \} returns the 3rd element stored in it

Moreover, this operator can also be applied to collections implementing Map interface e.g. HashMap etc.

- Key is specified inside brackets
- For example, \$\{ myMap["id"] \} returns the value associated with the id(key)

**EL – Robust Features**

Some powerful characteristics of Expression Language are:

- Multiple expressions can be combined and intermixed with static text. For example

  \$ \{ “Hello” \${user.firstName} \${user.lastName} \}

- EL also supports automatic type conversion; as a result primitive can implicitly wrap and unwrap into/from their corresponding java classes. For example

  \begin{equation}
  \text{begin} = \”\$\{ \text{student.marks} \}\”
  \end{equation}

- Most importantly, if object/identifier is null, no NullPointerException would be thrown. For example. If the expression written is:

  \$\{person.name\}

  Assume that person is null, then no exception would be thrown and the result would also be null.

**Using Expression Language**

Expression Language can be used in following situations

- As attribute values in standard & custom actions. E.g.

  \[
  \text{<jsp:setProperty}\text{ id = “person” value = $\{\ldots\} / >}
  \]
• In template text – the value of the expression is inserted into the current output. E.g.

```html
<h3> $ { ...... } </h3>
```

• With JSTL (discussed in the next handout)

**Example Code: AddressBook using EL**

So far, we have shown you implementation of AddressBook example in number of different ways. This time EL will be incorporated in this example. AddressBook code example consists on searchperson.jsp, showperson.jsp, ControllerServlet, PersonInfo and PersonDAO classes. Let’s look on the code of each of these components:

**PersonInfo.java**

The JavaBean used to represent one person record.

```java
package vu;
import java.io.*;
public class PersonInfo implements Serializable{
    private String name;
    private String address;
    private int phoneNum;

    // no argument constructor
    public PersonInfo() {
        name = "";
        address = "";
        phoneNum = 0;
    }

    // setters
    public void setName(String n) {
        name = n;
    }

    public void setAddress(String a) {
        address = a;
    }

    public void setPhoneNum(int pNo) {
        phoneNum = pNo;
    }

    // getters
```
public String getName() {
    return name;
}

public String getAddress() {
    return address;
}

public int getPhoneNum() {
    return phoneNum;
}
}

PersonDAO.java
It is used to retrieve/search person records from database.

class PersonDAO {
    private Connection con;

    // constructor
    public PersonDAO() throws ClassNotFoundException, SQLException {
        establishConnection();
    }

    // used to establish connection with database
    private void establishConnection() throws ClassNotFoundException, SQLException {
        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
        String conUrl = "jdbc:odbc:PersonDSN";
        con = DriverManager.getConnection(conUrl);
    }

    // used to search person records against name
    public ArrayList retrievePersonList(String pName) throws SQLException {
        ArrayList personList = new ArrayList();
        return personList;
    }
}
String sql = "SELECT * FROM Person WHERE name = ?";
    PreparedStatement pStmt = con.prepareStatement(sql);
    pStmt.setString(1, pName);
System.out.println("retrieve person list");
ResultSet rs = pStmt.executeQuery();
    String name;
    String add;
    int pNo;

    while (rs.next()) {
        name = rs.getString("name");
        add = rs.getString("address");
        pNo = rs.getInt("phoneNumber");

        // creating a PersonInfo object
        PersonInfo personBean = new PersonInfo();
        personBean.setName(name); personBean.setAddress(add);
        personBean.setPhoneNum(pNo);

        // adding a bean to arraylist
        personList.add(personBean);
    } // end while

    return personList;
} // end retrievePersonList

//overriding finalize method to release resources
public void finalize() {
    try{
        if(con != null){
            con.close();
        }
    }
    catch(SQLException sex) {
        System.out.println(sex);
    }
} // end finalize

} // end class
searchperson.jsp
This JSP is used to gather person's name from the user and submits this data to the ControllerServlet.

<html>
   <body>
      <center>
         <h2>Address Book</h2>
         <h3>Search Person</h3>
         <FORM name="search" action="controllservlet" />
         <TABLE BORDER="1">
            <TR>
               <TD> <h4>Name</h4>  </TD>
               <TD> <input type="text" name="name" /> </TD>
            </TR>
            <TR> <TD COLSPAN="2" ALIGN="CENTER">  
               <input type="submit" value="search" />
               <input type="reset" value="clear" />
            </TD> </TR>
         </TABLE>
         </FORM>
      </center>
   </body>
</html>

ControllerServlet.java
The Controller Servlet receives request from searchperson.jsp and after fetching search results from database, forwards the request to showperson.jsp.

package controller;
import vu.*;
import java.io.*;
import java.net.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ControllerServlet extends HttpServlet {
    // end processRequest()

    // This method only calls processRequest()
    protected void doGet(HttpServletRequest request,
                         HttpServletResponse response) throws ServletException, IOException {
        processRequest(request, response);
    }

    // This method only calls processRequest()
    protected void doPost(HttpServletRequest request,
                           HttpServletResponse response) throws ServletException, IOException {
        processRequest(request, response);
    }

    protected void processRequest(HttpServletRequest request,
                                   HttpServletResponse response) throws ServletException, IOException {
        // defined below
        searchPerson(request, response);
    }

    protected void searchPerson(HttpServletRequest request,
                                 HttpServletResponse response) throws ServletException, IOException {
        try {
            // creating PersonDAO object
            PersonDAO pDAO = new PersonDAO();

            // retrieving request parameter “name” entered on showperson.jsp
            String pName = request.getParameter("name");
        }
    }
}
// calling DAO method to retrieve personlist from database // against the name entered by the user
ArrayList personList = pDAO.retrievePersonList(pName);

// storing personlist in request scope, later it is retrieved // back on showperson.jsp
request.setAttribute("plist", personList);

// forwarding request to showperson, so it renders personlist
RequestDispatcher rd = request.getRequestDispatcher("showperson.jsp");
rд.forward(request, response);

}catch (Exception ex) {
    System.out.println("Exception is" + ex);
}

} // end searchPerson

showperson.jsp

This page is used to display the search results. To do so, it reclaims the stored ArrayList (personList) from the request scope. Furthermore, this page also uses the Expression Language to display records.

<%@ page import="java.util.*" %>
<%@ page import="vu.*" %>
<html>
<body>
<center>
<h2>Address Book</h2>
<h3>Following results meet your search criteria</h3>

<TABLE BORDER="1">
<TR>
<TH>Name</TH>
<TH>Address</TH>
<TH>Phone Num</TH>
</TR>
</TABLE>

<%-- start of scriptlet --%>
<%  
// retrieving ArrayList from request scope
ArrayList personList=(ArrayList)request.getAttribute("plist");
PersonInfo person = null;

for(int i=0; i<personList.size(); i++) {
    person=(PersonInfo)personList.get(i);
    // storing PersonInfo object in request scope /* As mentioned, an object must be
    // stored in some scope to work with Expression Language*/
    request.setAttribute("p",person);
}

<%-- end of scriptlet --%>

<TR>
<%-- accessing properties of stored PersonInfo object with name “p” using EL --%>
<TD>$ {p.name} </TD> <TD>$ {p.address} </TD>
<TD>$ {p.phoneNum}</TD>
<%-- The following expressions are now replaced by EL statements written above--%>

<%--<%= person.getName()%> --%> <%--<%= person.getAddress()%> --%> <%--<%= person.getPhoneNum()%> --%>
</TR>

<%  
} // end for  
%

</TABLE>

</center>

</body>

</html>
web.xml

<?xml version="1.0" encoding="UTF-8"?>

<web-app>
    
    <servlet>
        <servlet-name>ControllerServlet</servlet-name>
        <servlet-class>controller.ControllerServlet</servlet-class>
    </servlet>
    
    <servlet-mapping>
        <servlet-name>ControllerServlet</servlet-name>
        <url-pattern>/controllerservlet</url-pattern>
    </servlet-mapping>
    
</web-app>

References:

. Java A Lab Course by Umair Javed.
. Expression Language Tutorial by Sun

http://java.sun.com/j2ee/1.4/docs/tutorial/doc/JSPIntro7.html

The JSTL Expression Language by David M. Geary

http://www.informit.com/articles/article.asp?p=30946&rl=1
JavaServer Pages Standard Tag Library (JSTL)

Introduction

The JSP Standard Tag Library (JSTL) is a collection of custom tag libraries that implement general-purpose functionality common to Web applications, including iteration and conditionalization, data management formatting, manipulation of XML, and database access. Like JSP, JSTL is also a specification not an implementation. The development theme of JSTL is “scriptlet free JSP”.

These tag libraries provide a wide range of custom action functionality that most JSP authors have found themselves in need of in the past. Having a defined specification for how the functionality is implemented means that a page author can learn these custom actions once and then use and reuse them on all future products on all application containers that support the specification. Using the JSTL will not only make your JSPs more readable and maintainable, but will allow you to concentrate on good design and implementation practices in your pages.

JSTL & EL

JSTL includes supports for Expression Language thus EL can be used to specify dynamic attribute values for JSTL actions without using full-blown programming language. Prior to JSP 2.0, EL can only be used in attributes of JSTL tags but EL now becomes a standard part of JSP 2.0. This allows the use of EL anywhere in the document.

Functional Overview

As mentioned, JSTL encapsulates common functionality that a typical JSP author would encounter. This set of common functionality has come about through the input of the various members of the expert group. Since this expert group has a good cross-section of JSP authors and users, the actions provided in the JSTL should suit a wide audience. While the JSTL is commonly referred to as a single tag library, it is actually composed of four separate tag libraries:

- **Core** - contains tags for conditions, control flow and to access variables etc.
- XML manipulation - contains tags for XML parsing and processing
- **SQL** - contains tags for accessing and working with database.
- Internationalization and formatting - contains tags to support locale messages, text, numbers and date formation
Twin Tag Libraries

JSTL comes in two flavors to support various skill set personal

- **Expression Language (EL) version**
  - Dynamic attribute values of JSTL tags are specified using JSTL expression language (i.e. \${expression})
  - The EL based JSTL tag libraries along with URIs and preferred prefixes are given below in tabular format

<table>
<thead>
<tr>
<th>Library</th>
<th>URI</th>
<th>Prefix</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core</td>
<td><a href="http://java.sun.com/jsp/jstl/core">http://java.sun.com/jsp/jstl/core</a></td>
<td>c</td>
</tr>
<tr>
<td>SQL</td>
<td><a href="http://java.sun.com/jsp/jstl/sql">http://java.sun.com/jsp/jstl/sql</a></td>
<td>sql</td>
</tr>
<tr>
<td>Internationalization/Format</td>
<td><a href="http://java.sun.com/jsp/jstl/fmt">http://java.sun.com/jsp/jstl/fmt</a></td>
<td>fmt</td>
</tr>
<tr>
<td>XML</td>
<td><a href="http://java.sun.com/jsp/jstl/xml">http://java.sun.com/jsp/jstl/xml</a></td>
<td>x</td>
</tr>
</tbody>
</table>

- **Request Time (RT) version**
  - Dynamic attribute values of JSTL tags are specified using JSP expression (i.e. `<%= expression %>`)
  - The RT based JSTL tag libraries along with URIs and preferred prefixes are given below in tabular format

<table>
<thead>
<tr>
<th>Library</th>
<th>URI</th>
<th>Prefix</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core</td>
<td><a href="http://java.sun.com/jsp/jstl/core_rt">http://java.sun.com/jsp/jstl/core_rt</a></td>
<td>c_rt</td>
</tr>
<tr>
<td>SQL</td>
<td><a href="http://java.sun.com/jsp/jstl/sql_rt">http://java.sun.com/jsp/jstl/sql_rt</a></td>
<td>sql_rt</td>
</tr>
<tr>
<td>Internationalization/Format</td>
<td><a href="http://java.sun.com/jsp/jstl/fmt_rt">http://java.sun.com/jsp/jstl/fmt_rt</a></td>
<td>fmt_rt</td>
</tr>
<tr>
<td>XML</td>
<td><a href="http://java.sun.com/jsp/jstl/xml_rt">http://java.sun.com/jsp/jstl/xml_rt</a></td>
<td>x_rt</td>
</tr>
</tbody>
</table>

**Using JSTL**

As we discussed earlier, JSTL includes four standard tag libraries. As is true with any JSP custom tag library, a `taglib` directive must be included in any page that you want to be able to use this library's tags.

For example, to use EL based core tag library, the `taglib` directive appears as:

```jsp
<%@taglib prefix="c" uri=http://java.sun.com/jsp/jstl/core %>
```
And to use RT based core tag library, the taglib directive appears as:

```jsp
<%@taglib prefix="c_rt" uri="http://java.sun.com/jsp/jstl/core_rt"%
```

**Working with Core Actions (tags)**

The set of tags that are available in the Core tag library come into play for probably most anything you will be doing in your JSPs such as:

- Manipulation of scoped variables
- Output
- Conditional logic
- Loops
- URL manipulation
- and Handling errors.

Let’s walk through some important core actions:

**c:set**

Provides a tag based mechanism for creating and setting scope based variables. Its syntax is as follows:

```jsp
<c:set var="name" scope = "scope" value = "expression"/>
```

Where the var attribute specifies the name of the scoped variable, the scope attribute indicates which scope (page | request | session | application) the variable resides in, and the value attribute specifies the value to be bound to the variable. If the specified variable already exists, it will simply be assigned the indicated value. If not, a new scoped variable is created and initialized to that value.

The scope attribute is optional and default to page.

Three examples of using cset are given below. In the first example, a page scoped variable “timezone” is set to a value “Asia / Karachi”.

```jsp
<c:set var="timezone" value = "Asia/Karachi"/>
```

In the second example, a request scoped variable “email” email is set to a value “me@gmail.com”

```jsp
<c:set var="email" scope = "request" value = "me@gmail.com"/>
```

In the third example, a page scoped variable “email” is set to value of request parameter “email” by using param implicit object. If email parameter is defined in JSP page as:

```jsp
<input type="text" value = "email"/>
```
Then cset tag would be used as:

```xml
<c:set var="email" scope="request" value="param.email" />  
```

Using cset with JavaBeans & Map

c: set tag can also be used to change the property of a bean or the value against some key. For this purpose, the syntax of the cset tag would look like this:

```xml
<c:set target="bean/map" property="property/key" value="value" />
```

- If target is a bean, sets the value of the property specified. This process is equivalent to <jsp:setProperty … /> JSP action tag.
- If target is a Map, sets the value of the key specified

And of course, these beans and maps must be stored in some scope prior to any attempt is made to change their properties.

For example, consider the following snippet of code that stores PersonInfo’s object person into request scope using <jsp:useBean … /> tag. Then using cset tag, person’s name property is set to “ali”.

```xml
<jsp:useBean id="person" class="vu.PersonInfo" scope="request" />
<c:set target="person" property="name" value = "ali" />
```

c: out

A developer will often want to simply display the value of an expression, rather than store it. This can be done by using c:out core tag, the syntax of which appears below:

```xml
<c:out value = "expression" default = "expression" />
```

This tag evaluates the expression specified by its value attribute, and then prints the result. If the optional default attribute is specified, the c:out action will print its (default) value if the value attribute's expression evaluates either to null or an empty String. This tag is equivalent to JSP expression i.e. <%= expression %>. Consider the following examples in which the usage of c:out tag has shown. In the first example, string “Hello” would be displayed

```xml
c:out value = "Hello" />
```

In the second example, if request parameter num evaluates to null or an empty string then default value “0” would be displayed.

```xml
c:out value = "${param.num}" default = "0" />
The above fragment of code is equivalent to following scriptlet:

```jsp
<% 
    String no = request.getParameter("num");
    if (no == null || no.equals") 
       { 
         out.println(0);
       }
    else 
    { 
        out.println(no);
    }
%
```

If we want to display the property of a bean like name, we'll write

```jsp
<c:out value="${person.name}" default = "Not Set" />
```

### c:remove

As its name suggests, the `c:remove` action is used to delete a scoped variable, and takes two attributes. The `var` attribute names the variable to be removed, and the optional `scope` attribute indicates the scope from which it should be removed and defaults to page.

For example, to remove a variable named `square` from page scope, we'll write:

```jsp
<c:remove var = "square" />
```

And if variable `email` is required to be removed from request scope, then `c:remove` tag will look like:

```jsp
<c:remove var = "email" scope = "request" />
```

### c:forEach

In the context of Web applications, iteration is primarily used to fetch and display collections of data, typically in the form of a list or sequence of rows in a table. The primary JSTL action for implementing iterative content is the `c:forEach` core tag. This tag supports two different styles of iteration:

- Iteration over an integer range (like Java language's `for` statement)
- Iteration over a collection (like Java language's Iterator and Enumeration classes)
Iteration over an Integer range

To iterate over a range of integers, the syntax of the c:forEach tag will look like:

```
<c:forEach var="name" begin="expression" end="expression" step="expression">
    Body Content
</c:forEach>
```

The begin and end attributes should be either constant integer values or expressions evaluating to integer values. They specify the initial value of the index for the iteration and the index value at which iteration should cease, respectively. When iterating over a range of integers using c:forEach, these two attributes are required and all others are optional.

The step attribute specifies the amount to be added to the index after each iteration. Thus the index of the iteration starts at the value of the begin attribute, is incremented by the value of the step attribute, and halts iteration when it exceeds the value of the end attribute. Note that if the step attribute is omitted, the step size defaults to 1.

If the var attribute is specified, then a scoped variable with the indicated name will be created and assigned the current value of the index for each pass through the iteration. This scoped variable has nested visibility that is it can only be accessed within the body of the c:forEach tag.

For example to generate squares corresponding to range of integer values, the c:forEach tag will be used as:

```
<c:forEach var="x" begin="0" end="10" step="2">
    <c:out value="${x * x}" />
</c:forEach>
```

By executing the above code, following output would appear: 4 16 36 64 100

Iteration over a Collection

When iterating over the members of a collection and arrays etc, one additional attribute of the c:forEach tag is used: the items attribute. Now the c:forEach tag will look similar to this:

```
<c:forEach var="name" items="expression">
    Body Content
</c:forEach>
```
When you use this form of the c:forEach tag, the items attribute is the only required attribute. The value of the items attribute should be the collection/array over whose members the iteration is to occur, and is typically specified using an EL expression. If a variable name is also specified using var attribute, then the named variable will be bound to successive elements of the collection for each iteration pass.

For example, to iterate over a String array (messages) using Java code, we used to write in JSP:

```jsp
<% for(int i=0; i<messages.length; i++) {
    String msg = messages[i];
%>
<%=
%>
<%=
<% 
} // end for 
%>
```

This can be done using c:forEach tag in much simpler way as shown below:

```
<c:forEach var="msg" items="${messages}" >
    <c:out value="${msg}" />
</c:forEach>
```

Similarly, to iterate over a persons ArrayList that contains PersonInfo objects, we used to write in JSP:

```jsp
<%
    ArrayList persons = (ArrayList)request.getAttribute("pList");
    for(int i=0; i<persons.size(); i++)
    {
        PersonInfo p == (PersonInfo)persons.get(i);
        String name = p.getName();
%>
    <%= name %>
    <%
} // end for 
%>
```
Indeed, the above task can be achieved in much simpler way using c:forEach tag as shown below:

```xml
<c:forEach var="p" items="${persons}" />
<c:forEach>

The c:forEach tag processes each element of this list(persons) in turn, assigning it to a scoped variable named p. Note that typecast is also not required.

Furthermore, you can use the begin, end, and step attributes to restrict which elements of the collection are included in the iteration.

**c:if**

Like ordinary Java’s if, used to conditionally process the body content. It simply evaluates a single test expression and then processes its body content only if that expression evaluates to true. If not, the tag’s body content is ignored. The syntax for writing c:if tag is:

```xml
<c:if test="expression" />

Body Content
</c:if>
```

For example, to display a message “a equals b” if two strings a & b are equal, the c:if tag is used as:

```xml
<c:if test="${a == b}" />
<h2>A equals B</h2>
</c:if>
```

**c:choose**

c:choose the second conditionalization tag, used in cases in which mutually exclusively test are required to determine what content should be displayed. The syntax is shown below:

```xml
<c:choose>
    <c:when test="expression" />

    Body content

    </c:when>

    .................
</c:choose>
```
<c:otherwise>
  Body content
</c:otherwise>
</c:choose>

Each condition to be tested is represented by a corresponding <c:when> tag, of which there must be at least one. Only the body content of the first <c:when> tag whose test evaluates to true will be processed. If none of the <c:when> tests return true, then the body content of the <c:otherwise> tag will be processed.

Note, though, that the <c:otherwise> tag is optional; a <c:choose> tag can have at most one nested <c:otherwise> tag. If all <c:when> tests are false and no <c:otherwise> action is present, then no <c:choose> body content will be processed.

The example code given below illustrates the usage of <c:choose> tag in which two strings a & b are compared and appropriate messages are displayed:

<c:choose>
  <c:when test="a == b">
    <h2>a equals b</h2>
  </c:when>
  <c:when test="a <= b">
    <h2>a is less than b</h2>
  </c:when>
  <c:otherwise>
    <h2>Don’t know what a equals to</h2>
  </c:otherwise>
</c:choose>

**netBeans 4.1 and JSTL**

If you are using netBeans 4.1 IDE then you have to add JSTL library to your project manually. To do so, right click on the libraries folder, you can find it under project’s name and select the Add Library option. This is also shown in the following figure:
The Add Library dialog box opens in front of you. Select JSTL 1.1 option and press Add Library button. Now you can refer to any JSTL library in your JSPs.
Note: Remember that the JSTL 1.1 library is only added to current project. You have to repeat this step for each project in which you want to incorporate JSTL.

**Example Code: AddressBook using JSTL core tags**

This is the modified version of AddressBook that was built using Expression Language in the last handout. Only showperson.jsp is modified to incorporate JSTL core tags along with Expression Language in place of scriptlets. The remaining participants searchperson.jsp, ControllerServlet, PersonInfo and PersonDAO left unchanged. Let’s look on the code of each of these components:

**PersonInfo.java**

The JavaBean used to represent one person record.

```java
package vu;

import java.io.*;

public class PersonInfo implements Serializable {
    private String name;
    private String address;
    private int phoneNum;
    // no argument constructor
    public PersonInfo() {
        name = "";
        address = "";
        phoneNum = 0;
    }
    // setters
    public void setName(String n) {
        name = n;
    }
    public void setAddress(String a) {
        address = a;
    }
}
```

© Copyright Virtual University of Pakistan
public void setPhoneNum(int pNo) {
    phoneNum = pNo;
}

// getters
public String getName() {
    return name;
}

public String getAddress() {
    return address;
}

public int getPhoneNum() { return phoneNum;
}
Client Side Validation

&

JavaServer Faces (JSF)

In this handout, we'll talk about client side validation and also learn about growing in demand Java technology i.e. JSF. First start with client side validation

Client Side Validation

Forms validation on the client-side is essential --it saves time and bandwidth, and gives you more options to point out to the user where they've gone wrong in filling out the form. Furthermore, the browser doesn't have to make a round-trip to the server to perform routine client-side tasks. For example, you wouldn't want to send the browser to the server to validate that all of the required fields on a form were filled out. Any scripting language can be used to achieve the said objective. However, JavaScript and VBScript are two popular options

Why is Client Side Validation Good?

There are two good reasons to use client-side validation:

- It's a fast form of validation: if something's wrong, the alarm is triggered upon submission of the form.

- You can safely display only one error at a time and focus on the wrong field, to help ensure that the user correctly fills in all the details you need.

Code Example: Form Validation using JavaScript

For example on the following form, we want to make sure that text filed for name should not be left empty and age field does not contain any negative value. To accomplish this we'll use JavaScript.

If user forgets to provide name and/or enters a negative value, a message would be displayed to the user that indicates what was went wrong? However, if user conforms to requirements, he/she would be taken to another page that displays a greeting message.

Note: In this example, JavaScript semantics isn't discussed over here as I am assuming that you might be familiar with some scripting language. Otherwise, www.w3schools.com is an excellent resource to learn about scripting languages
The code that is used to generate this page is given below:

```html
<HTML>
<HEAD>
<!— start of scripting code and mentioning type -->

<SCRIPT TYPE = "text/javascript">
<!— defining a function that receives form’s reference, defined inside the body and returns false if any requirement violated -->

function validateForm(thisform) {

<!— checking the value of the name field, if it is left empty then displaying a message -->
if (thisform.name.value == null || thisform.name.value == "")
{
    alert("Username is required");
    return false;
}

<!— if value of age is negative, displaying a message -->
```
if (thisform.age.value < 0 )
{
    alert("Age can't be negative");
    return false;
}
} // end of function

</SCRIPT> <!--end of script-->

</HEAD>

<!-- validateForm method is called and specified as a value of onsubmit value, if this method returns false, the user remains on the same page -->

<FORM method="post" onsubmit="return validateForm(this)" action = "greeting.jsp" >
<h2>Client Side Validation Example </h2>
<br/>
Name: <INPUT type="text" name="name" size="30" />
<br/>
<br/> Age: <INPUT type="text" name="age" size="30" />
<br/> <br/> <br/> <br/>
</FORM>
</BODY>
</HTML>
JavaServer Faces (JSF)

JSF technology simplifies building the user interface for web applications. It does this by providing a higher-level framework for working with your web applications. Some distinct features will be discussed provided by this technology. To begin with, have a look on some popular existing frameworks

Different existing frameworks
- **Struts**
  
  A popular open source JSP-based Web application framework helps in defining a structured programming model (MVC), also validation framework and reduces tedious coding but…
  - Adds complexity and doesn’t provide UI tags
  - Very Java programmer centric

- **Tapestry**

  Another popular framework that is extensively used in the industry is Tapestry. It has almost similar sort of problems as with Struts.

JavaServer Faces
- **A framework which provides solutions for:**
  - Representing UI components
  - Managing their state
  - Handling events
  - Input validation
  - Data binding
  - Automatic conversion
  - Defining page navigation
  - Supporting internationalization and accessibility.

If you are familiar with Struts and Swing (the standard Java user interface framework for desktop applications), think of JavaServer Faces as a combination of those two frameworks. Like Swing, JSF provides a rich component model that eases event handling and component rendering; and like Struts, JSF provides Web application lifecycle management through a controller servlet

JSF UI Components

Some of the standard JavaServer Faces components are shown below:
Some custom JavaServer Faces components are:

- A custom component that allows users to select multiple options from a list.
- A custom component that displays a dynamic list of items based on user input.
- A custom component that integrates with social media platforms for sharing.
- A custom component that provides real-time updating of data from a server.

And some open course JavaServer Faces components are also available like:

<table>
<thead>
<tr>
<th>ID</th>
<th>First Name</th>
<th>Family Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Hans</td>
<td>Mueller</td>
</tr>
<tr>
<td>5</td>
<td>Amadeus</td>
<td>Mozart</td>
</tr>
<tr>
<td>6</td>
<td>Harry</td>
<td>Potter</td>
</tr>
</tbody>
</table>
And some third-party JavaServer Faces components are also available:

A JSF application works by processing events triggered by the JSF components on the pages. These events are caused by user actions. For example, when the user clicks a button, the button triggers an event. You, the JSF programmer, decide what the JSF application will do when a particular event is fired. You do this by writing event listeners. In other words, a JSF application is event-driven.

For example, if you write a JSF code to create a button, you will write:

```xml
<h:commandButton value="Login"
   actionListener="#{customer.loginActionListener}"
   action="#{customer.login}" />
```

The value attribute specifies the text that appeared on the face of a button, the actionListener attributes specifies to call the loginActionListener method written somewhere in a Customer class if an event is triggered and on which to go next, is decided by the login method of Customer class and given as a value of action attribute. The method specified in action attribute should return a String value as the returned Stringvalue is used in page navigation.

**Note:** Many IDE provides visual support for JSF so you can drag and drop components instead of writing tedious coding for defining JSF components as shown above. Sun Studio Creator® is a free open source IDE that provides visual support for JSF and can be downloaded form Sun site. The code examples are also built using this IDE.

A code snippet of a Customer class is given below:

```java
class Customer {

    public void loginActionListener(ActionEvent e) {
        ...................
    }

    public String login() {
        return "OK";
    }
}
```
Example Code: Hello User

The example code (“hello user 1”) is given along with the handout. It is strongly advised that you must see the lecture video in order to learn how this example is built. User will provide a name in the text field and his/her name after appending “hello” to it, would be displayed on the same page.

Validators make input validation simple and save developers hours of programming. JSF provides a set of validator classes for validating input values entered into input components. Alternatively, you can write your own validator if none of the standard validators suits your needs. Some built-in validators are:

- **DoubleRangeValidator**
  Any numeric type, between specified maximum and minimum values

- **LongRangeValidator**
  Any numeric type convertible to long, between specified maximum and minimum values

- **LengthValidator**
  Ensures that the length of a component's local value falls into a certain range (between minimum & maximum). The value must be of String type.

Example Code: Hello User

The example code (“hello user 2”) is given along with the handout. You can open it using Sun Studio Creator IDE. It is strongly advised that you must see the lecture video in order to learn how this example is built.

It is actually a modified version of the last example. This time, we’ll make sure that user couldn’t left blank the name field and must enter a name between ranges of 2 to 10 characters. If any condition fails, an appropriate message would be displayed.

**JSF – Managed Bean-Intro**

These are JavaBeans defined in the configuration file and are used to hold the data from JSF components. Managed beans represent the data model, and are passed between business logic and pages. Some other salient features are:

- Use the declarative model
- Entry point into the model and event handlers
- Can have beans with various states

Here is an example of a managed-bean element whose scope is session, meaning that an instance of this bean is created at the beginning of a user session.

```xml
<managed-bean>
  <managed-bean-name>myBean</managed-bean-name>
  <managed-bean-class>myPackage.MyBean</managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
```
JSF – Value Binding
Value binding expressions can be used inside of JSF components to:
- Automatically instantiate a JavaBean and place it in the request or session scope.
- Override the JavaBean's default values through its accessor methods.
- Quickly retrieve Map, List, and array contents from a JavaBean.
- Synchronize form contents with value objects across a number of requests.

The syntax of binding expressions is based on the JavaServer Pages (JSP) 2.0 Expression Language. In JSP, expressions are delimited with "${}" , but in JSF they are delimited with "#{ }".

JSF – Method Binding
Unlike a value binding, a method binding does not represent an accessor method. Instead, a method binding represents an activation method.

For example, binding an event handler to a method

```html
<h:commandButton ……
  
  actionListener="#{customer.loginActionListener}"

  .......... />
```

JSF Navigation
Page navigation determines the control flow of a Web application. JSF provides a default navigational handler and this behavior can be configured in configuration. However, you can do it visually in most tools like Sun Studio Creator

Note: We have quickly breezed through the JSF technology essentials due to shortage of time. You must explore it by yourself to excel on it. You can find the resources in the last handout to acquire further skills.

References:
- Java A Lab Course by Umair Javed
- Introduction to JavaServer Faces by Sun http://java.sun.com
- JavaServer Faces Programming by Kumiawan
Lesson 45

**JavaServer Faces**
In the last lecture, we have covered the basic nuts and bolts of JSF. Having a belief on “learning by doing”, in this lecture another example is also given to show you the capabilities of JSF.

**Example Code: Addition of Two Numbers**
The example code (“AddNumbers”) is given along with the handout. It is strongly advised that you must see the lecture video in order to learn the making plus working of this example. This example demonstrates the usage of value and method binding expressions, managed beans, and how to use page navigation technique using IDE etc.

**Web Services**
In the remaining handout, we’ll take an overview of web services’ potential, their types and working model. Resources are given at the end for those who are interested in learning new technologies.

**Introduction**
Web services are Web-based enterprise applications that use open, XML-based standards and transport protocols to exchange data with calling clients. Web Service is becoming one of those overused buzzwords these days. Due to their increasing popularity, Java platform Enterprise Edition (J2EE) provides the APIs and tools you need to create and deploy interoperable web services and clients.

**Web service, Definition by W3C**
W3C recently has come up with a decent definition of web services. According to W3C, “A Web service is a software application identified by a URI, whose interfaces and binding are capable of being defined, described and discovered by XML artifacts and supports direct interactions with other software applications using XML based messages via internet-based protocols”.

**Distributed Computing Evolution**
Let’s think a little bit on how distributed computing technology has evolved.

In the beginning, things were built and deployed typically in the form of client and server model in which clients talk to a single server, for example, remote procedure calls (RPC).
The second phase can be called web-based computing in which many clients talk to many servers through the net. In this phase, communicating partners still have to go through some pre-arrangement in terms of what common object model they have to use or what common communication protocol they have to agree upon. Finally, the web services model in which service users and service providers can be dynamically connected. And the pretty much every computing device and application participates as both service user and service provider.

**Characteristics of Web services**

Web services are XML-based throughout. Pretty much everything in the domain of Web services is defined in XML. For example, the format of the data being exchanged between service user and service provider is defined in XML, or the description of web service is defined in XML. Because the only contract that has to be agreed upon between service user and service provider is syntax and semantics of XML messages, as long as valid messages can be generated and understood, it does not matter what programming language is used. So a web service is said to be programming language independent.

Web services can be dynamically located and invoked. And typically they will be accessed and invoked over both internet and intranet.

- **Interoperable**
  Connect across heterogeneous networks using ubiquitous web-based standards
- **Economical**
  Recycle components, no installation and tight integration of software
- **Automatic**
  No human intervention required even for highly complex transactions
- **Accessible**
  Legacy assets & internal apps are exposed and accessible on the web
- **Available**
  Services on any device, anywhere, anytime
- **Scalable**
  No limits on scope of applications and amount of heterogeneous applications

**Types of Web service**

- **Data providers**
  For example, a service providing stock quotes
- **Business-to-business process integration**
  For example, purchase orders
- **Enterprise application integration**
  Different applications work together simply by adding a webservice wrapper
Comparison between Web page & Web service
Just to give you a sense on the difference between a web page and a web service, consider the following table:

<table>
<thead>
<tr>
<th>Web page</th>
<th>Web Service</th>
</tr>
</thead>
<tbody>
<tr>
<td>Has a UI</td>
<td>No GUI</td>
</tr>
<tr>
<td>Interacts with user</td>
<td>Interacts with application</td>
</tr>
<tr>
<td>Works with web browser client</td>
<td>Works with any type of client</td>
</tr>
</tbody>
</table>

Web service Architectural Components
Following are the core building blocks of web service architecture.

- **Service Description** – how do clients know how it works (which functions, parameters etc.)?
  At the minimum, you need a standard way of describing a web service that is universally understood by all potential service users and service providers. This is important because without commonly agreed upon description of service, a service provider might have to produce individually tailored way of describing its service to all its potential service users.

  Web Service Description Language (WSDL pronounced as viz-dal) is industry agreed upon XML language that can be used to describe web service. It provides XML format for describing web services in terms of methods, properties, data types and protocols.

- **Service Registration (Publication) and Discovery** There has to be registry by which a service can be published and discovered.

  Universal Description, Discovery & Integration (UDDI), a way to publish and find web services. A repository of web services on the internet where a machine or a human can find different web services.
  www.uddi.org

- **Service Invocation** Then there has to be standard way of invoking a service. Finally, for business transactions in which secure and reliable message delivery is important, there has to be a standard electronic business framework.

The following figure represents simplified web service architecture and summarizes the working of web services:
References:

- Java A Lab Course by Umair Javed
- Web services overview by sang shin

Resources:

An excellent resource for learning Java related technologies is:

http://www.apl.jhu.edu/~hall/java/

- http://java.sun.com
- http://www.javaworld.com
- http://www.theserverside.com
- http://www.jsfcentral.com
- http://www.jspolympus.com
- http://www.onjava.com