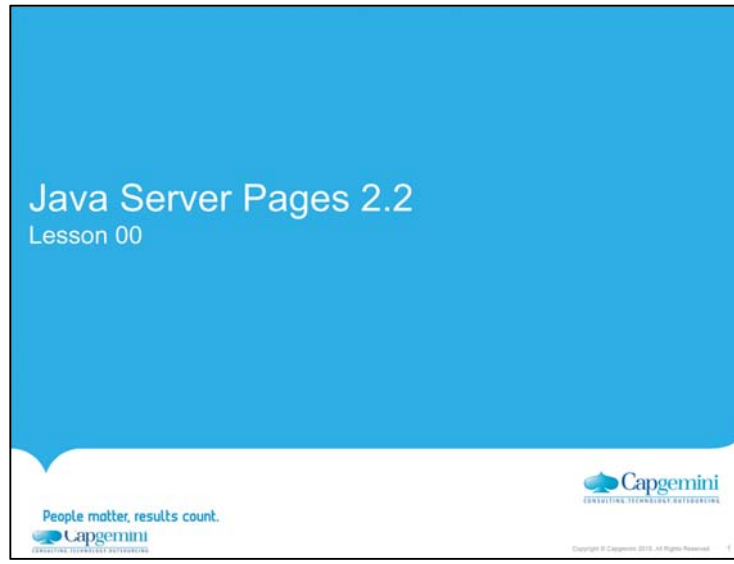


Document History

Date	Course Version No.	Software Version No.	Developer / SME	Change Record Remarks
Nov – 2009	2.0	2.1	Habib & Mahima	Revamped from JSP 2.0 to JSP 2.1
23-Nov-2009			CLS Team	Review
Feb – 2010	2.1	2.1	Mahima	Refined based on the feedback received from faculties
May – 2011	2.2	2.1	iLearn	Refinement.
July --- 2013	2.3	2.1	Mohan Chinnaiah	Refinement.
Feb--2015	2.4	2.2	Anjulata	Revamped from JSP2.1 to JSP2.2
May-2016	2.6	2.2	Anjulata	Revamped the course for 2 days

Capgemini

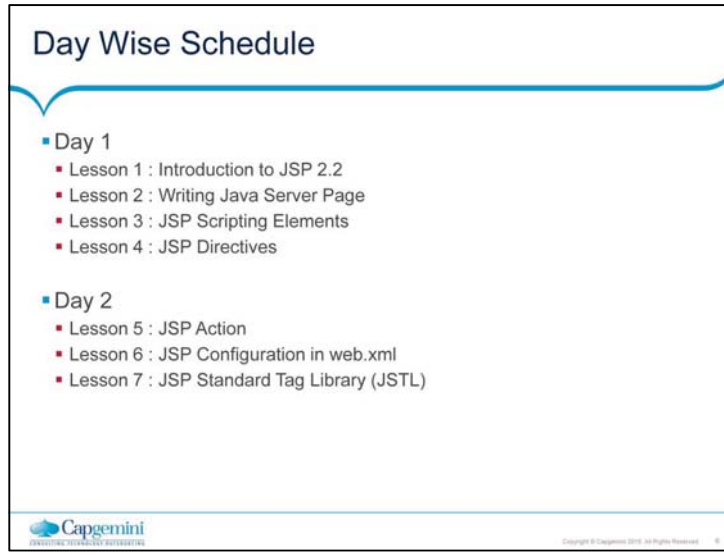

Course Goals and Non Goals
Course Goals
Developing dynamic web pages
Developing web based applications
Integrating Servlets, JSPs & Java Beans
Course Non Goals
Developing distributed Web based applications/Enterprise application
Capgemini


Pre-requisites
Java
DBMS/SQL
HTML
Javascript
XML
Servlet
Capgemini

Intended Audience

- Web Application Authors
- Web Application Developers/Programmers

Capgemini


Day Wise Schedule
Day 1
Lesson 1 : Introduction to JSP 2.2
Lesson 2 : Writing Java Server Page
Lesson 3 : JSP Scripting Elements
Lesson 4 : JSP Directives
Day 2
Lesson 5 : JSP Action
Lesson 6 : JSP Configuration in web.xml
Lesson 7 : JSP Standard Tag Library (JSTL)
Capgemini

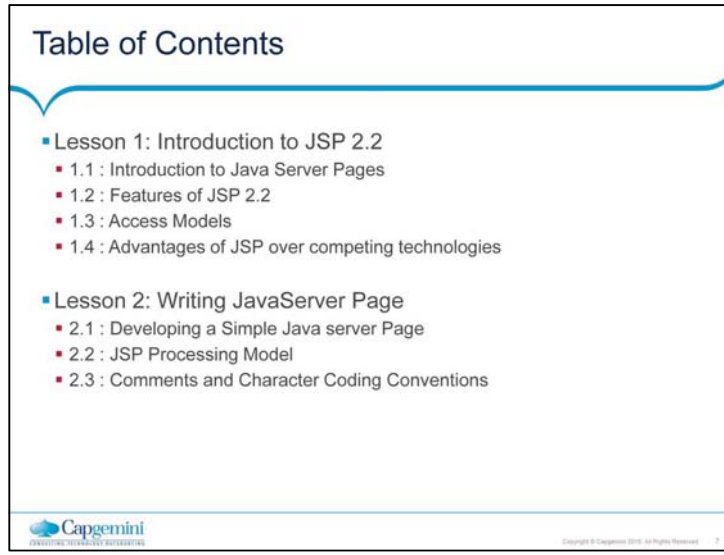


Table of Contents
Lesson 1: Introduction to JSP 2.2
1.1 : Introduction to Java Server Pages
1.2 : Features of JSP 2.2
1.3 : Access Models
1.4 : Advantages of JSP over competing technologies
Lesson 2: Writing JavaServer Page
2.1 : Developing a Simple Java server Page
2.2 : JSP Processing Model
2.3 : Comments and Character Coding Conventions
Capgemini

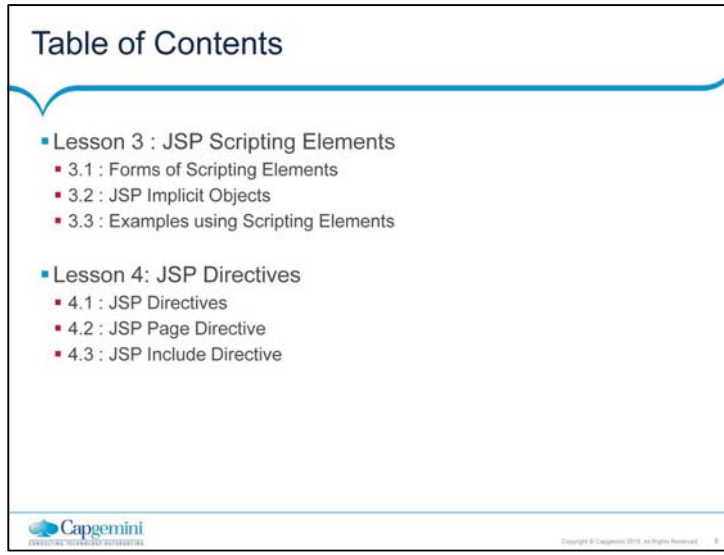


Table of Contents
Lesson 3 : JSP Scripting Elements
3.1 : Forms of Scripting Elements
3.2 : JSP Implicit Objects
3.3 : Examples using Scripting Elements
Lesson 4: JSP Directives
4.1 : JSP Directives
4.2 : JSP Page Directive
4.3 : JSP Include Directive
Capgemini

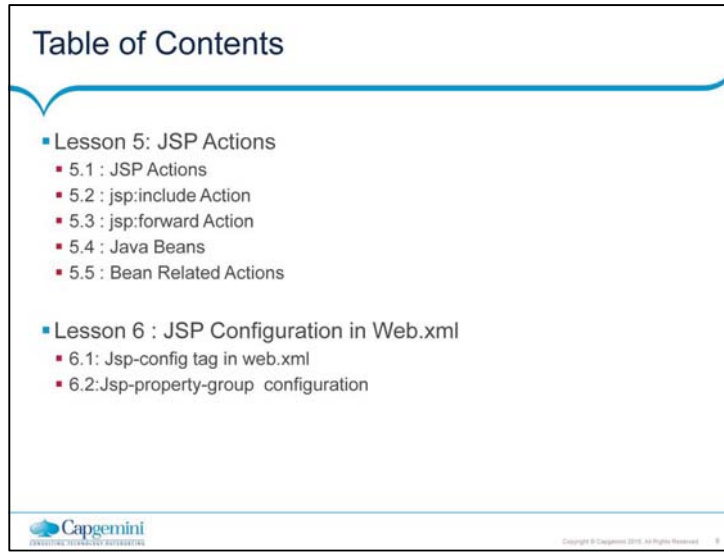


Table of Contents
Lesson 5: JSP Actions
5.1 : JSP Actions
5.2 : jsp:include Action
5.3 : jsp:forward Action
5.4 : Java Beans
5.5 : Bean Related Actions
Lesson 6 : JSP Configuration in Web.xml
6.1: Jsp-config tag in web.xml
6.2:Jsp-property-group configuration
Capgemini

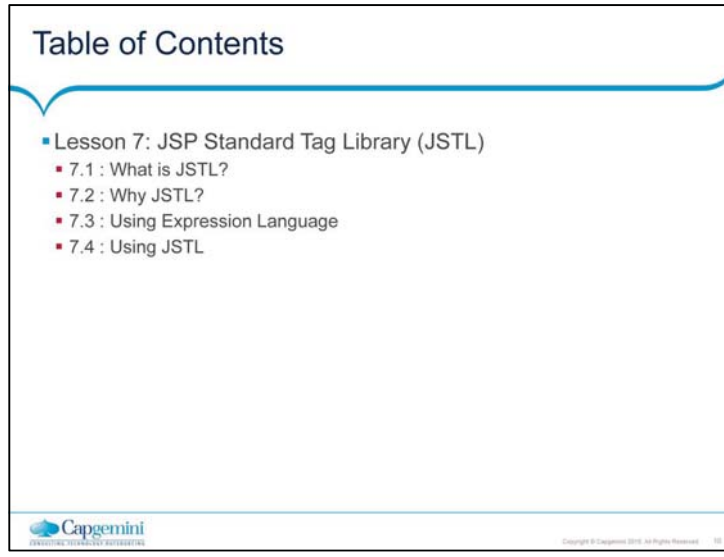


Table of Contents
Lesson 7: JSP Standard Tag Library (JSTL)
7.1 : What is JSTL?
7.2 : Why JSTL?
7.3 : Using Expression Language
7.4 : Using JSTL
Capgemini

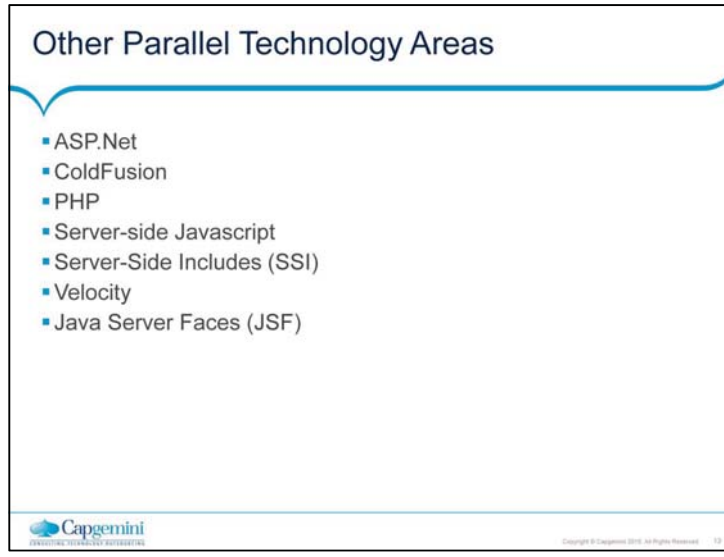
References

- Java Server Pages for Beginners; by Sharanam & Vaishali Shah, Cynthia Bayross
- Core Servlets & JSP; by Marty Hall
- JSP 2.2 specification download site :
  - http://download.oracle.com/otndocs/jcp/jsp-2.2-mrel-oth-JSpec
- JSP 2.2 on-line documentation :
  - http://java.sun.com/products/jsp/2.2/docs/jsp-2_2-pfd2/index.html

Capgemini

Next Step Courses

- Apache Struts Framework


Other Parallel Technology Areas
ASP.Net
ColdFusion
PHP
Server-side Javascript
Server-Side Includes (SSI)
Velocity
Java Server Faces (JSF)
Capgemini

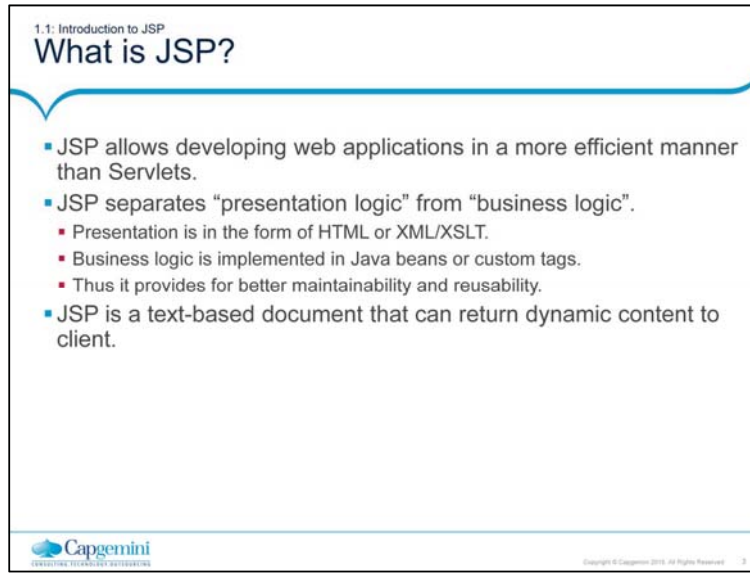
Java Server Pages (JSP)

Lesson 01: Introduction to JSP 2.2

Lesson Objectives

- In this lesson, you will learn:
  - Introduction to Java Server Pages (JSP)
  - Features of JSP 2.2
  - Access Models
    - JSP Model 1 Architecture
    - JSP Model 2 Architecture
  - Advantages of JSP over competing technologies

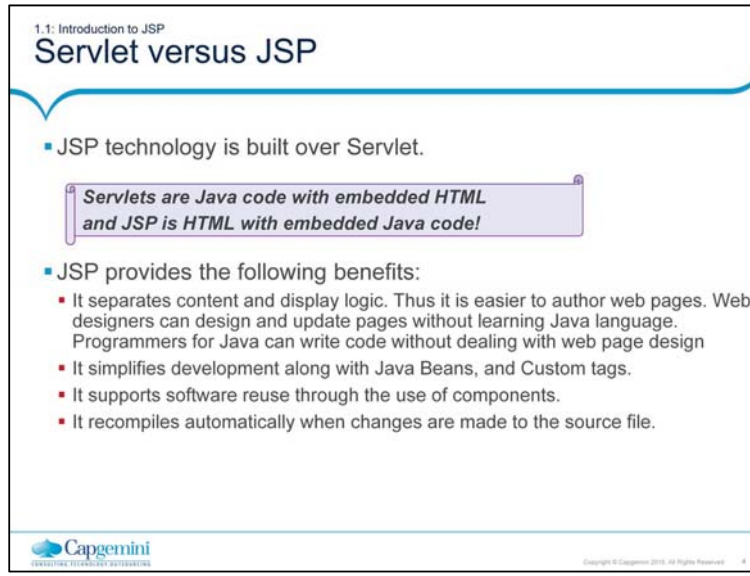
Capgemini



Introduction to JSP:

What is JSP?

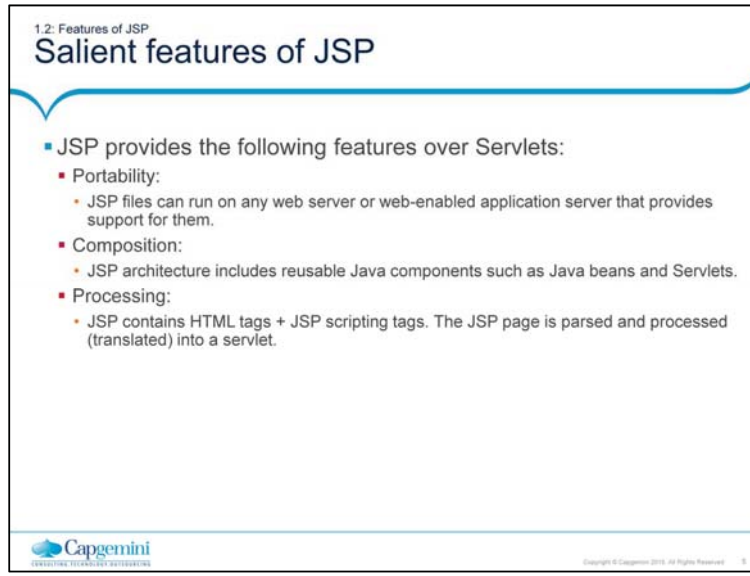
- **Java Server Pages (JSP)** is a simple, yet powerful technology for creating and maintaining dynamic-content webs pages. Based on the Java programming language, JSP offers proven portability, open standards, and a mature re-usable component model.
- JSP was introduced as a follow-on technology to Servlets.
  - **Downside of Servlet:** Servlet needs to provide many print statements to generate HTML response. Any change to the presentation means recompilation and re-deployment!
- JSP separates "presentation logic" from "business logic" and thus enables the Web page designer and Web Application developer to work independently! Besides, the business logic is encapsulated in Java beans or custom tags. Hence the use of JSP increases **reusability of code**, as well.
- JSP is a **text-based document** that can return dynamic content to client. It contains both "**static**" and "**dynamic content**".



Introduction to JSP:

JSP versus Servlet:

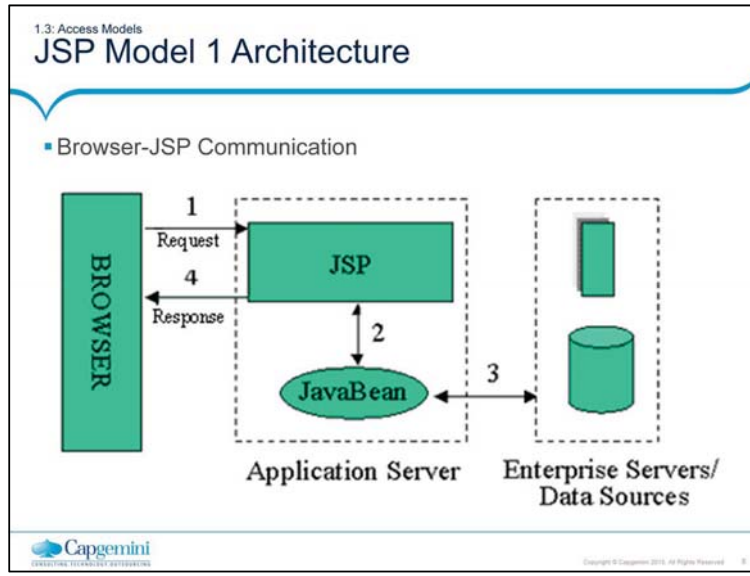
- **JSP** technology is built over **Servlet**. In fact, all JSP pages when requested, are first converted into Servlet. Thus they inherit all benefits of Servlets.
- Although dynamic, interactive pages with Servlets alone can be created using JSP technology makes the process even easier.
  - With JSP pages, it is easy to combine “static templates” with code to generate “dynamic content”.
  - JSP page structure makes it easier to author pages “by hand”.
  - JSP provides easy XML-like tags to invoke Java beans, thus hiding business complexity from page authors.
- JSP allows to clearly separate **content** from **presentation**. This is unlike Servlets where business logic and presentation logic co-exist many times in a single program! By separating the look from the content, different people can work on different tasks. Web page design experts can build the HTML, leaving places for Servlet programmers to insert the dynamic content.



Features of JSP Pages:

JSP provides the following features:

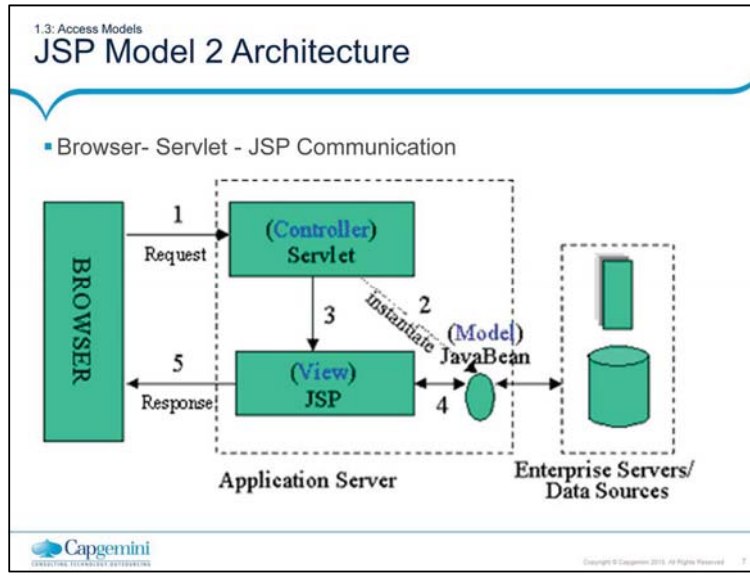
- **Portability:** JSPs files can be run on any web server or web-enabled application server that provides support for them. This support involves recognition, translation, and management of the Java Server Page lifecycle and its interactions with associated components. As long as the server, on which the Java Server Pages executes, supports the same specification level as that to which the file was written, no changes should be necessary as files are moved from server to server. However, instructions for the setup and configuration of the files may differ between files.
- **Composition:** The Java Server Pages architecture includes reusable Java components. The architecture also allows embedding a scripting language (default language for now is Java) directly into the Java Server Pages file. The components currently supported include JavaBeans and Servlets. Support for Enterprise Java Beans components are likely be added in a future release.
- **Processing:** A JSP file is essentially an HTML document with JSP scripting or tags. The Java Server Pages file has a *.jsp* extension. Before the page is served, the JSP syntax is parsed and processed into a servlet on the server side. The servlet that is generated outputs real content in straight HTML for responding to the client. Since it is standard HTML, the dynamically generated response looks no different to the client browser than a static response.



Access Methods:

JSP Model 1 Architecture:

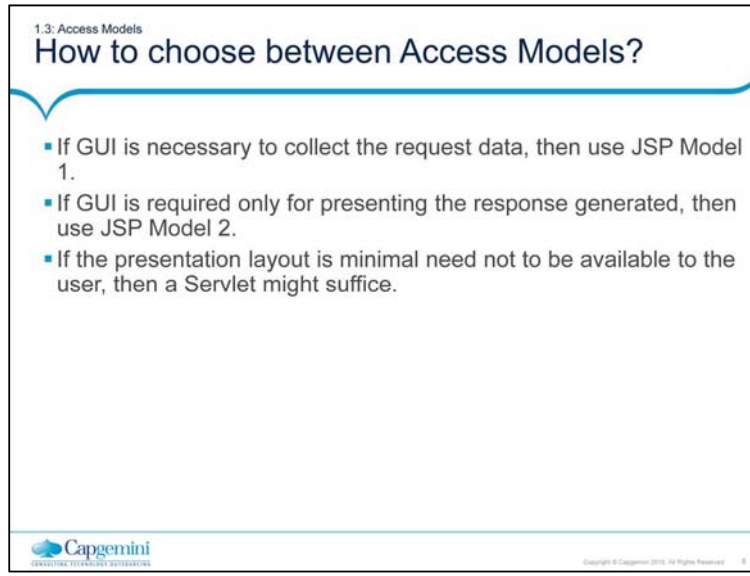
- In this model, the client request comes directly to a Java Server Page.
- Suppose the page accesses reusable JavaBean components (for example: JavaBeans) that perform particular well-defined computations like accessing a database. Then the result of the Bean's computations, called **result sets** are stored within the Bean as properties. The page uses such beans to generate dynamic content and present it back to the client.



Access Models:

JSP Model 2 Architecture:

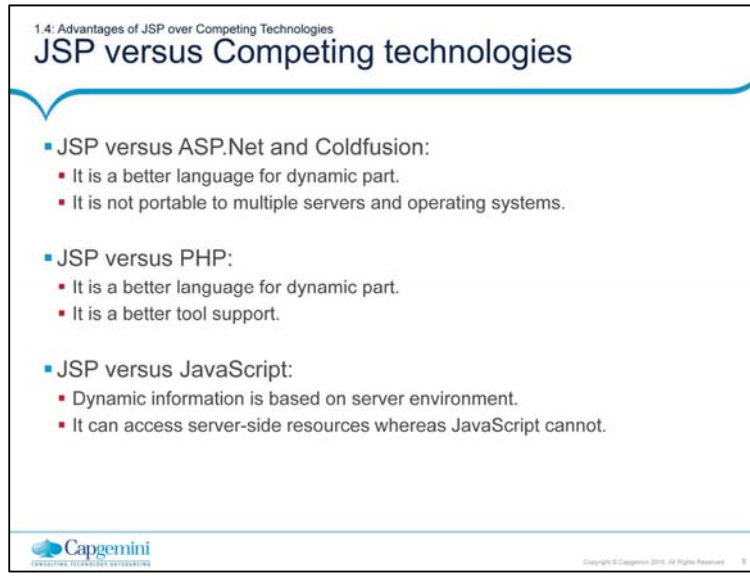
- In this model, the request comes through a servlet. The servlet generates the dynamic content. To handle the response to the client, the servlet creates a Bean and stores the dynamic content (sometimes called the **result set**) in the Bean. The servlet then invokes a Java Server Page that will present the content generated by the Bean.
- In this model, the servlet acts as a ***controller*** responsible for processing requests and creating any beans needed by the JSP page. The controller is also responsible for deciding to which JSP page to forward the request. The JSP page retrieves objects created by the servlet and extracts dynamic content for insertion within a template. This model promotes the use of the **Model View Controller (MVC)** architectural pattern.
- There are two APIs to support this model of request processing using Java Server Pages.
  - One API facilitates passing context between the invoking servlet and the Java Server Page.
  - The other API lets the invoking servlet specify which Java Server Page to use.
- In both the above cases, the page can contain any valid Java code, as well. The Java Server Pages architecture encourages separation of content from presentation, however it does not mandate it.



Access Models:

How to choose between Access Models?

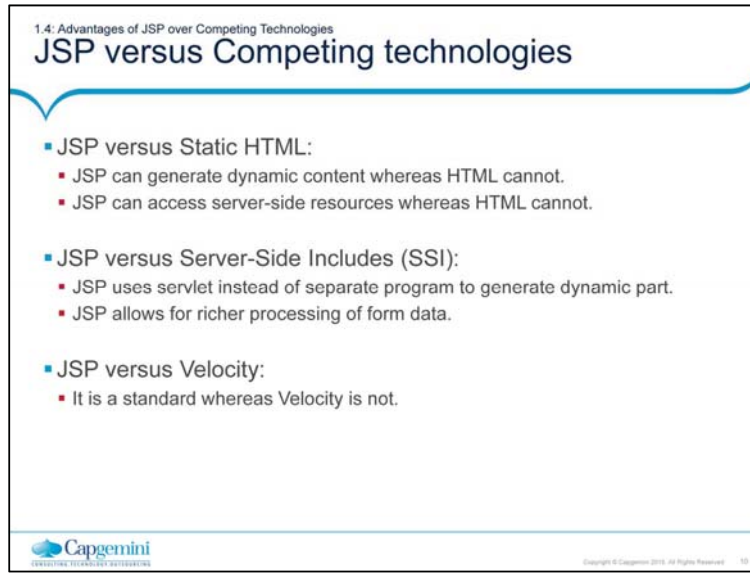
- With at least two access models, the question naturally arises "When does it make sense to have a Java Server Page as the front-end to a servlet, as the back-end, or use only the servlet?" Here are some possible guidelines:
  - If a graphical interface (GUI) is necessary to collect the request data, then use a Java Server Pages file.
  - If the request and request parameters are otherwise available to the servlet, but the results of the servlet processing requires a graphical interface to present them, then use a Java Server Pages file.
  - If presentation layout is minimal (will not require many println lines in servlet code) and does not require to make that presentation logic available to a customer or webpage designer, then a Servlet might sufficient.



Advantages of JSP over Competing Technologies:

JSP versus Competing technologies:

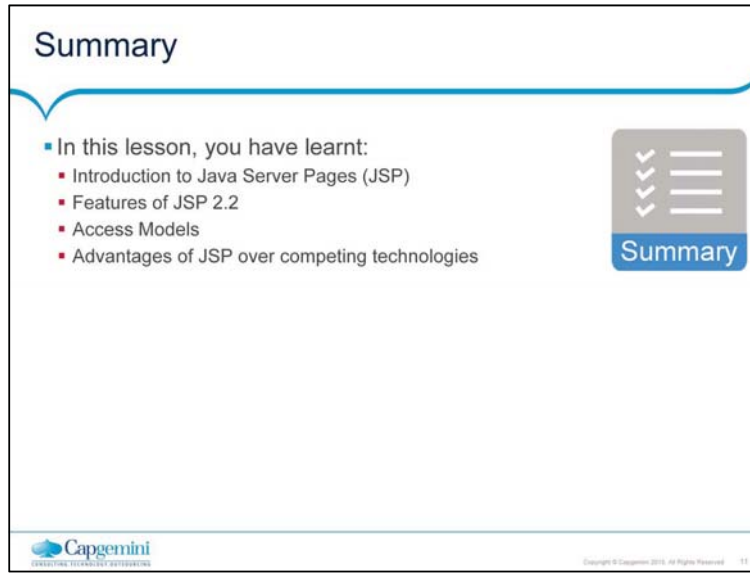
1. **JSP versus ASP.Net and Coldfusion:** ASP.Net is a Microsoft technology. Hence the dynamic part is written in Microsoft-specific language whereas in JSP it is written in Java, so it is powerful and promotes portability. Coldfusion also uses a proprietary language hence not portable. Secondly, JSP is portable to multiple servers and operating systems whereas ASP.Net and Coldfusion are not.
2. **JSP versus PHP:** The PHP is a scripting language which again not portable across servers. Unlike for JSP, the tool support for PHP is not very great.
3. **JSP versus JavaScript:** JavaScript can generate dynamic html on the client. However, the dynamic information is based on the client's environment. With the exception of cookies, form submission data is not available to JavaScript. Additionally, since JavaScript runs on client, it cannot access server-side resources.



Advantages of JSP over Competing Technologies:

JSP versus Competing technologies:

4. **JSP versus Static HTML:** HTML cannot generate dynamic information. Also similar to Javascript it cannot access server-side resources.
5. **JSP versus Server-Side Includes (SSI):** SSI is a widely supported technology for including externally defined pieces into a static web page. JSP is better because it allows to use Servlets instead of a separate program to generate that dynamic part. Besides, SSI is intended for only simple inclusions, and not for "real" programs that use form data, make database connections, and so on.
6. **JSP versus Velocity:** JSP allows separation of application logic from presentation but it does not mandate it. Due to this, it is very easy for the developers to mix up these two. Whereas, Velocity enforces the Model2 architecture by enforcing separation between the application logic and presentation layout logic. However, it is not a standard.


Summary
In this lesson, you have learnt:
Introduction to Java Server Pages (JSP)
Features of JSP 2.2
Access Models
Advantages of JSP over competing technologies
Summary
Capgemini

Review – Questions

- Question 1: JSP architecture enables the separation of content generation from content ___.
- Question 2: The servlet resulting from the JSP uses the ___ method to process the client request.
- Question 3: In JSP Model 1 architecture, the client request is processed by ___.

Capgemini

Review – Questions

- Question 4: JSP Model 2 enforces ___ architectural pattern.
- Question 5: The advantage of JSP over ASP.Net is ___ and ___.

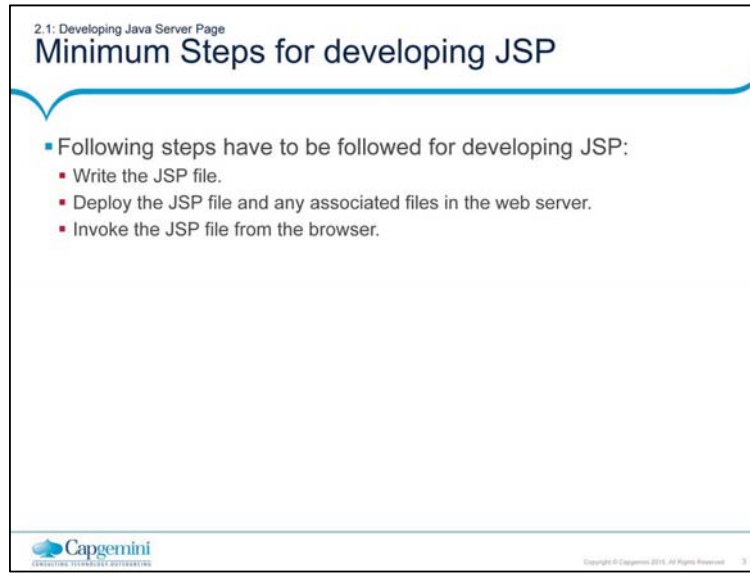
Java Server Pages (JSP)

Lesson 02: Writing Java Server Page

Lesson Objectives

- In this lesson, you will learn:
  - Developing a Java Server Page
  - JSP Processing Model
    - JSP Lifecycle
  - Comments and Character Quoting Conventions
    - Output Comment
    - Hidden Comment

Capgemini

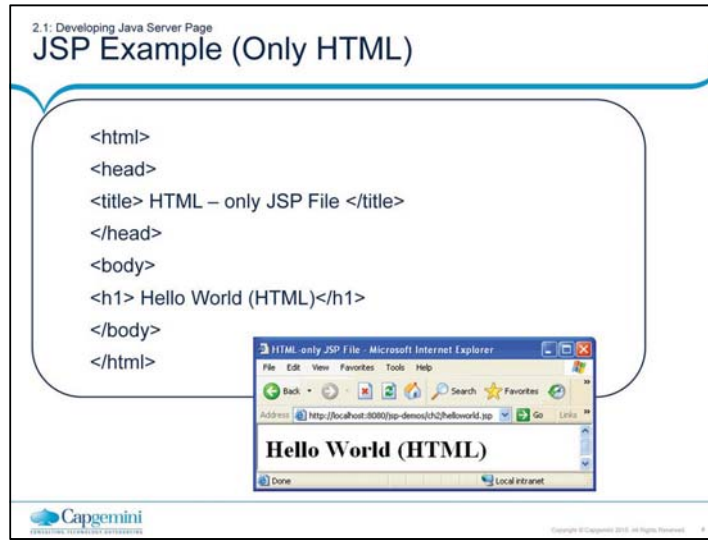


Developing Java Server Page:

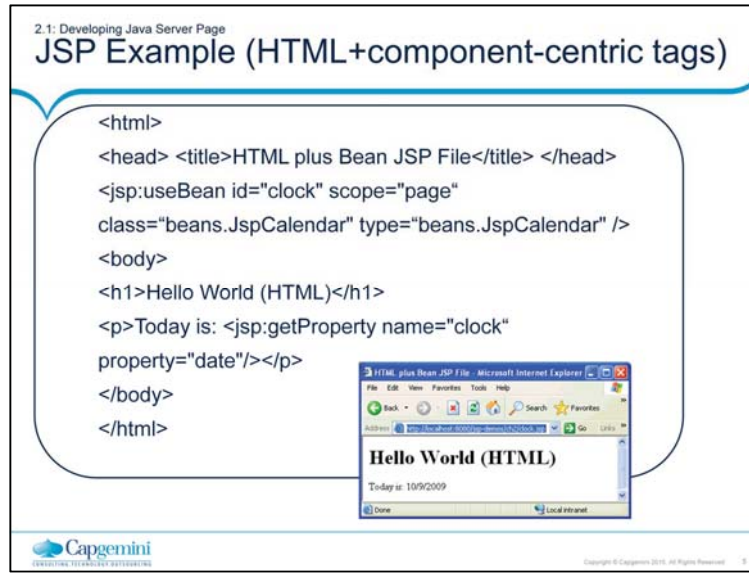
Minimum steps needed to develop a Java Server Page:

Following steps have to be followed for developing JSP:

1. Write the JSP File
   a) Declare any JavaBeans components.
   b) Use **tag-centric syntax** to access Bean properties or **scripting-centric syntax** to provide desired functionality.
   c) Save the file with a **.jsp** filename extension.
2. Deploy the JSP file and any associated files in the web server (Please refer to the JSP lab book for more details).
   a) Place the **.jsp** files under the context root directory.
   b) Place associated **.class** files in **WEB-INF\classes** folder and **.jar** files in **web-inf\ lib** folder.
3. Invoke the JSP file from a web browser.
   a) Type the URL in the browser as follows:
   http://<host-name>/<context-root>/<path>/<jsp file name>



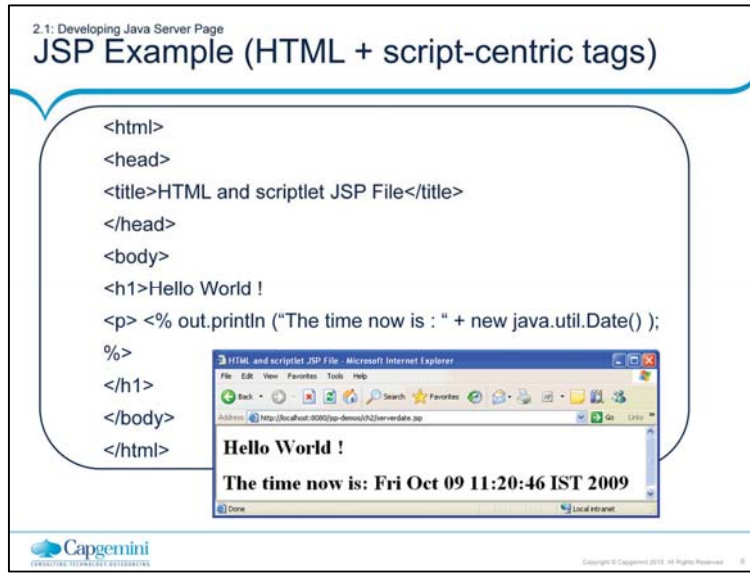
Developing Java Server Page:
The above slide shows an example of a JSP file that contains only HTML.
A file containing nothing but standard HTML code can be renamed with a filename extension of .jsp and therefore meet the minimum requirements to be invoked as a Java Server Page. Such a file would still be parsed and compiled to a servlet. The servlet would still return a response when the page was invoked - its response to the client would simply contain the original HTML.



Developing Java Server Page:

JSP Example (HTML + component-centric tags):

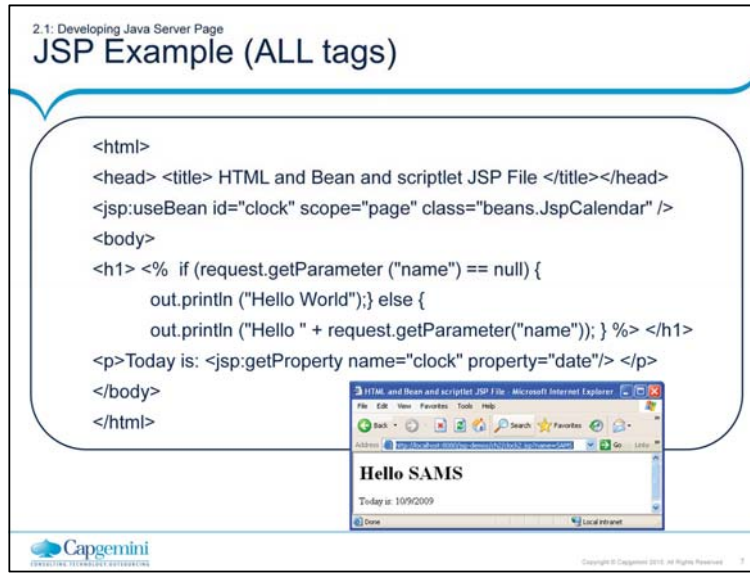
- The above slide shows an example of a JSP file that contains **HTML** and **component-centric tags**.
- Of course, the whole point of Java Server Pages technology is to simply and easily manage dynamic content on the server side. So, the plain HTML file from above, we can add some Java Server Pages tags to interact with Java components. Component can be as simple as a Bean, which returns the current time.



Developing Java Server Page:

JSP Example (HTML + script-centric tags)

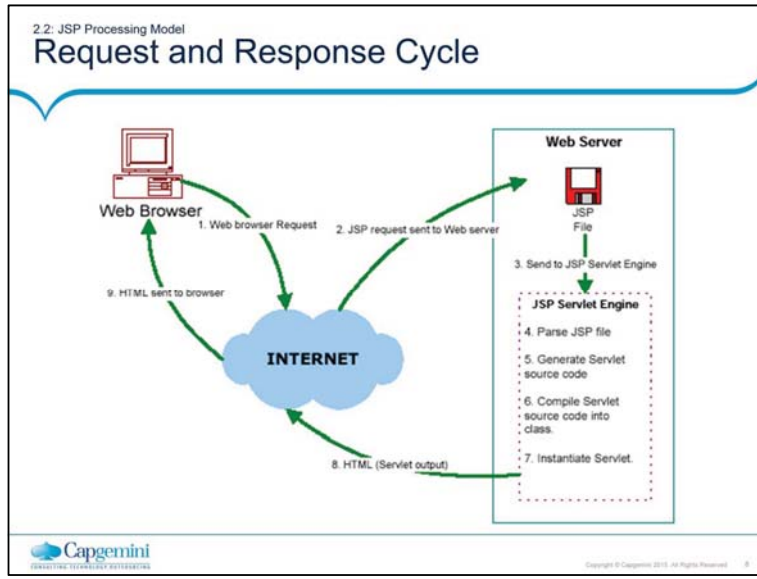
- The above slide shows an example of a JSP file that contains **HTML** and **script-centric tags**.
- Although Java Server Pages architecture encourages the use of **componentization** for ease of maintenance and reusability, It does not required to use components. This example uses a simple **scriptlet** containing raw Java code.



Developing Java Server Page:

JSP Example (All tags):

- The above slide shows an example of a JSP file that contains **HTML, component centric** and **script-centric tags**.
  This is the same example where we had only HTML tags with the substitution of two lines of code for the original Hello line. By checking for a name parameter in the incoming request (available in the automatic request object), the page can return either a generic or a personalized greeting.
- Remember to save the file with a **.jsp** filename extension. This tells the web server that the file is a Java Server Pages file and to process it accordingly.

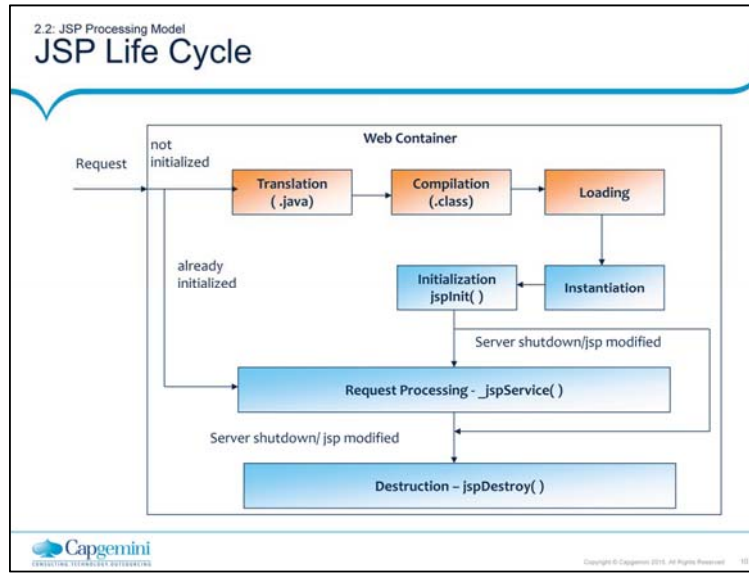

2.2: JSP Processing Model
Request and Response Cycle
Web Server
Web Browser
1. Web browser Request
2. JSP request sent to Web server
JSP
File
3. Send to JSP Servlet Engine
9. HTML sent to browser
JSP Servlet Engine
4. Parse JSP file
INTERNET
5. Generate Servlet source code
6. Compile Servlet source code into class.
7. Instantiate Servlet.
8. HTML (Servlet output)
Capgemini

2.2: JSP Processing Model

JSP Life Cycle

Phase Name	Description
Page translation	The page is parsed and a Java file containing the corresponding servlet is created.
Page compilation	The Java file is compiled.
Load class	The compiled class is loaded.
Create instance	An instance of the servlet is created.
Call jspInit()	This method is called before any other method to allow initialization.
Call _jspService()	This method is called for each request.
Call jspDestroy()	This method is called when the servlet container decides to take the servlet out of service.

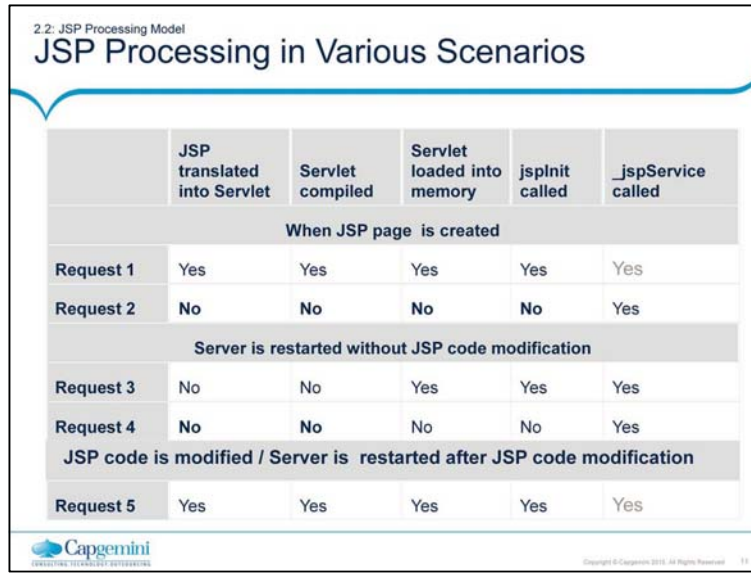
Capgemini



JSP Processing Model:

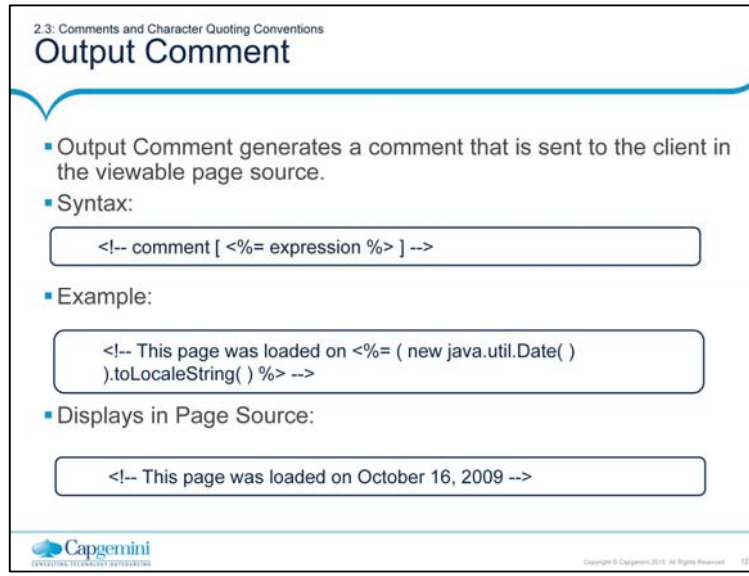
JSP Life Cycle:

- As mentioned earlier, JSP is stored as text file (.jsp) in the web application. When the client requests for the page for the first time, the server (web container) translates the JSP into **Java source code** (.java) and if there are no translation errors then it is compiled into a **servlet class** file. The servlet class is then loaded by the class loader, and then instantiated. For the first time request, the **jspInit( )** method (if present) is called to initialize resources.
- If the request for the JSP page is not the first time request, then the translation to Initialization steps are skipped and the request is directly processed by the **_jspService( )** method.
- The web container invokes the **jspDestroy** method (if present), to cleanup resources, when the server shuts down or if the JSP page is replaced with a modified one.
- The lifecycle methods of the JSP page translated into the servlet have the following form:
  - public void jspInit ( )
  - public void jspDestroy( )
  - public void _jspService (HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException
- The .java and .class file generated from the JSP page have the same name as the jsp file with a postfix "**_jsp**".
  For example: If the jsp file name is **helloworld.jsp**, then the generated files have the names as **helloworld_jsp.java** and **helloworld_jsp.class**.



JSP Processing in Various Scenarios:

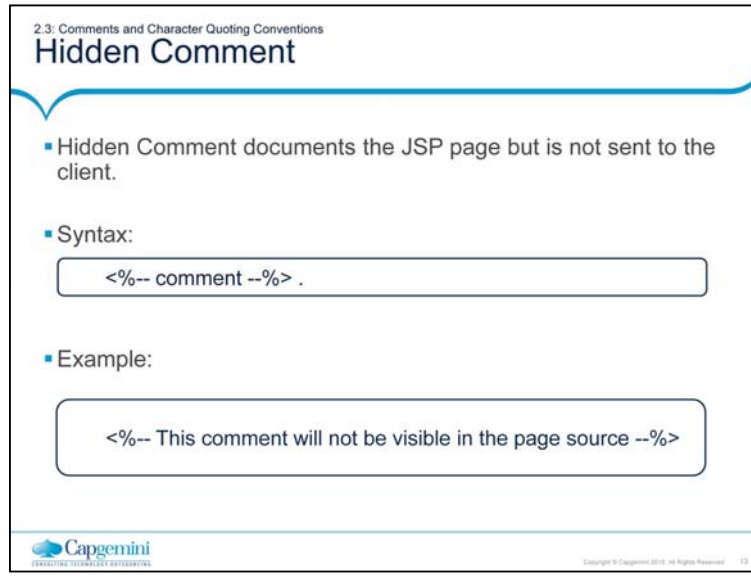
- The table shown on the above slide gives some common scenarios and tells us how a JSP is processed in each of these scenarios.
- Note that the servlets resulting from JSP pages use the **_jspService** method and not **doGet()** or **doPost()** methods.



Comments and Character Quoting Conventions:

Output Comment:

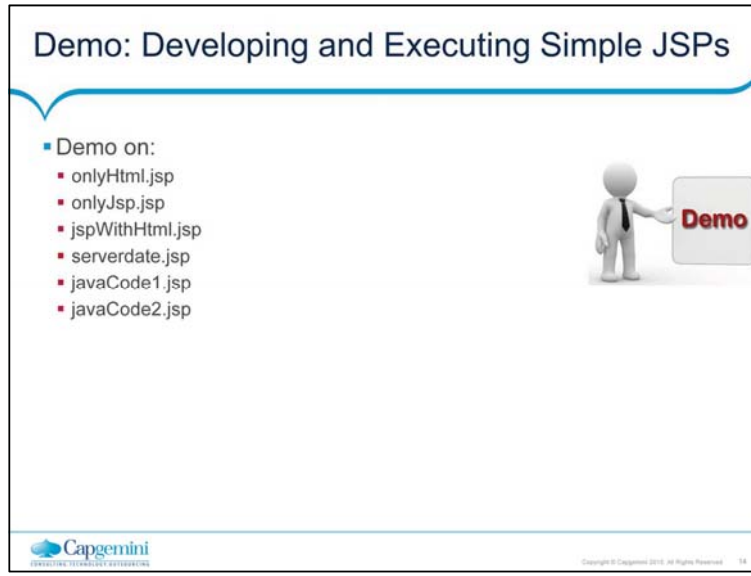
- The **JSP engine** handles an **output comment** as **non-interpreted HTML text**, returning the comment in the HTML output sent to the client. User can see the comment by viewing the page source from Web browser.
- An **Expression** included in a comment is dynamic. It is evaluated when the Web browser loads the page (that is, when the user first loads the page or reloads it later). So, any valid JSP expression can be used here.



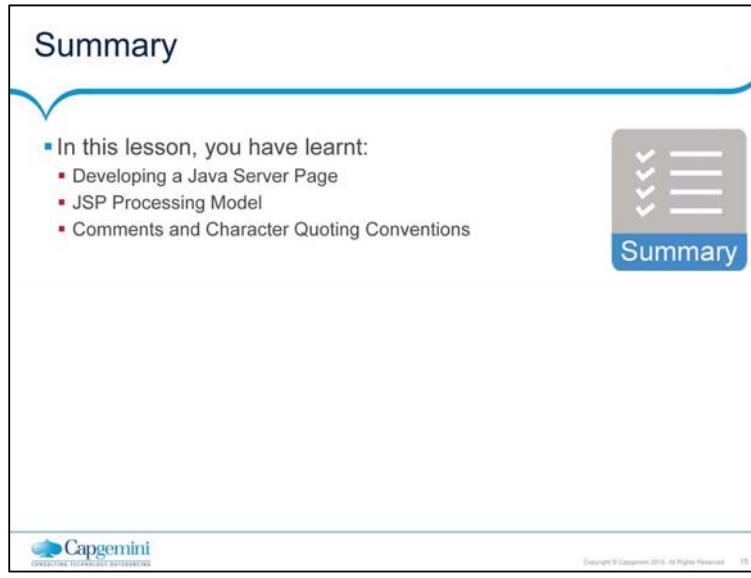
Comments and Character Quoting Conventions:

Hidden Comment:

- The JSP engine ignores a hidden comment, and does not process any code within hidden comment tags. A hidden comment is not sent to the client, either in the displayed JSP page or the HTML page source. The hidden comment is useful when we want to hide or "comment out" part of JSP page.
- Any characters in the body of the comment except the closing --%> combination can be used. If --%> is used in comment, it can be escaped by typing --%\>.



Deploy web application **Lesson2-JSPIntroduction** and show demo by executing each of the above JSP pages.


Summary
In this lesson, you have learnt:
Developing a Java Server Page
JSP Processing Model
Comments and Character Quoting Conventions
Summary
Capgemini

Review – Questions

- Question 1: A JSP page can consist of HTML,___, and ___ centric tags.
- Question 2: JSP pages should be placed under ___ directory of the web application.
- Question 3: ___ lifecycle method is called only once in the lifetime of JSP.

Capgemini

Review – Questions

- Question 4: ___ lifecycle method of JSP processes the client request.
- Question 5: <%-- comment --%> is a ___ comment.

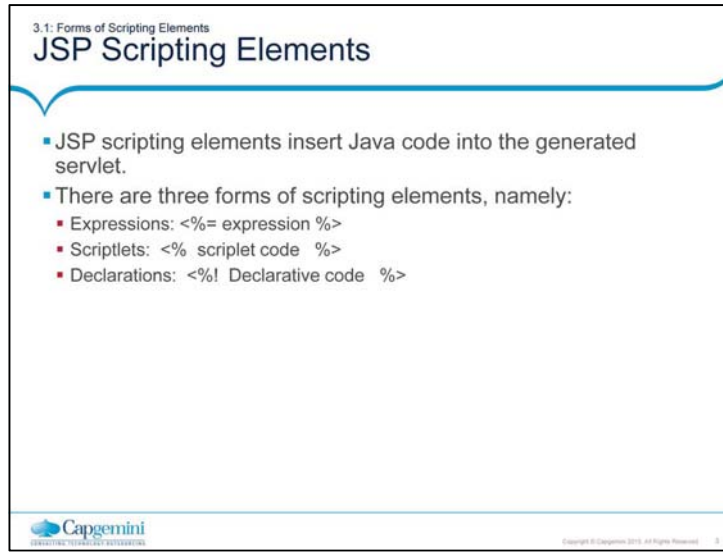
Java Server Pages (JSP)

Lesson 03: JSP Scripting Elements

Lesson Objectives

- In this lesson, you will learn about:
  - Forms of Scripting Elements
    - JSP Expressions
    - JSP Scriptlets
    - JSP Declarations
  - JSP Implicit Objects
  - Examples using Scripting Elements

Capgemini

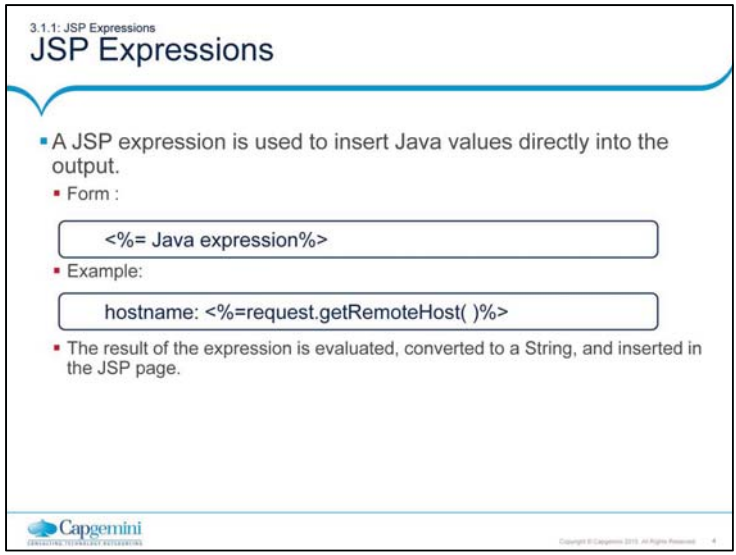


Forms of Scripting Elements:

JSP Scripting Elements:

- With JSP scripting elements Java code into the servlet that will be generated from the current JSP page.
- There are three forms of scripting elements:
  - **Expressions** of the form <%= *expression* %> that are evaluated and inserted into the output.
  - **Scriptlets** of the form <% *code* %> that are inserted into the servlet's service method.
  - **Declarations** of the form <%! *code* %> that are inserted into the body of the servlet class, outside of any existing methods.

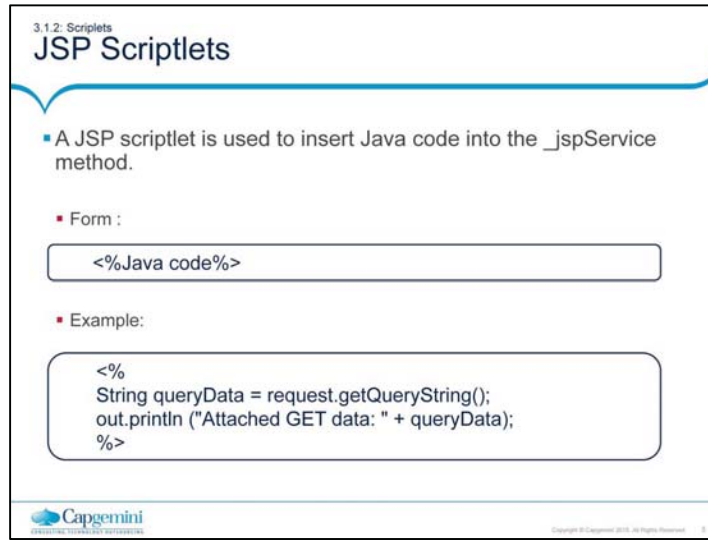
- Each of these are described in more detail in the subsequent slides.



Forms of Scripting Elements:

JSP Expressions:

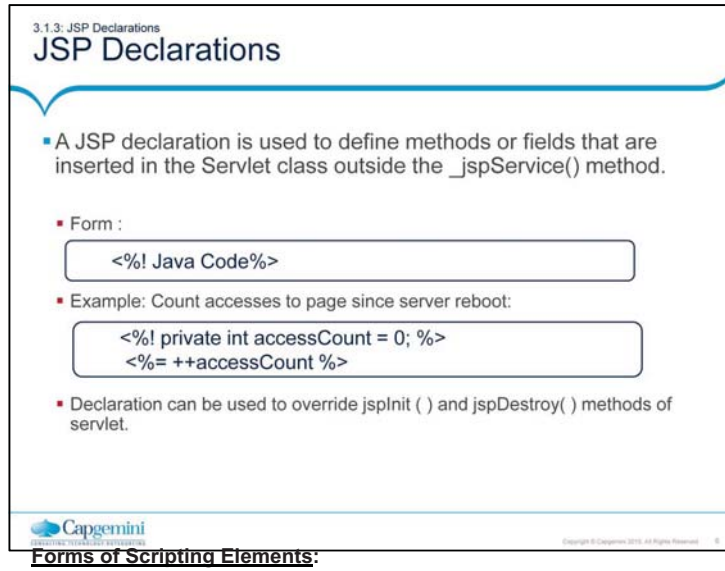
- The **JSP expression** is evaluated at run-time (when the page is requested), and thus has full access to information about the request. The JSP expression example shown on the above slide will retrieve the remote host information from the pre-defined **request object** and insert it in the **JSP** page. The result is in fact stored in the **out object** and inserted where the **expression** appears in the JSP page.
- To simplify these expressions, there are a number of predefined variables that can be used. These implicit objects are discussed in more detail later, but for the purpose of expressions, the most important ones are as follows:
  - **request:** the HttpServletRequest;
  - **response:** the HttpServletResponse;
  - **session:** the HttpSession associated with the request (if any); and
  - **out:** the PrintWriter (a buffered version of type JspWriter) used to send output to the client.
- When using Java as the scripting language, remember that:
  - A semicolon should not be used to end an expression.
  - The expression tag can contain any expression that is valid according to the Java Language Specification.
  - Expressions are evaluated in left-to-right order as they appear in the tag.



Forms of Scripting Elements:
JSP Scriptlets:
Scriptlets have the following form: <% Java Code %>. Scriptlets are executed when the JSP engine processes the client request. If the scriptlet produces output, then the output is stored in the out object, from which it can be displayed.
Note that code inside a scriptlet gets inserted exactly as written. Any static HTML (template text) before or after a scriptlet gets converted to print statements. This means that scriptlets need not contain complete Java statements, and blocks left open can affect the static HTML situated outside of the scriptlets.
For example:
Consider the following JSP fragment, containing mixed template text and scriptlets:

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

```
if (Math.random() < 0.5) {
  out.println("Have a <B>nice</B> day!");
} else {
  out.println("Have a <B>lousy</B> day!");
}
```

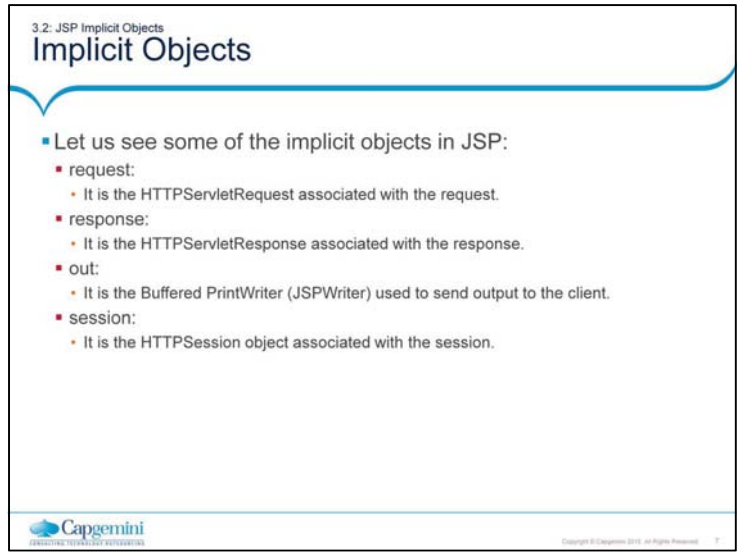


Forms of Scripting Elements:

JSP Declarations:

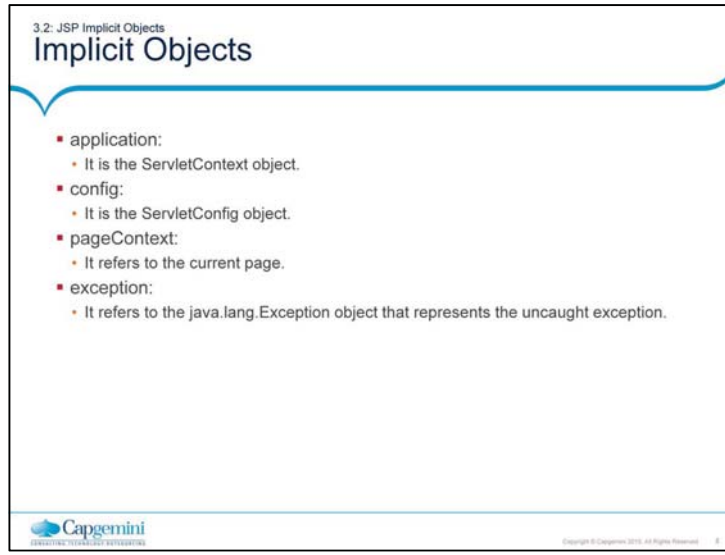
- A JSP *declaration* lets helps to define methods or fields that are inserted in the servlet class (outside of the service method processing the request).
- It has the following form: ***<%! Java Code %>***
- The example, shown on the slide prints out the number of times the current page has been requested since the server booted (or the servlet class was changed and reloaded). The **accessCount** variable is now an instance variable that is defined only once within the lifecycle of JSP. If, however, **accessCount** was part of a scriptlet, then it would have been a local variable whose scope would be the **service()** method!
- The JSP usually runs as multiple "**threads"** of one single instance. Different threads interfere with **variable access**, because it will be the same variable for all of them. If variables has to be used in JSP, it should be used with "**synchronized access"**, but that hurts the performance. In general, any data should go either in the "**session object**" or the "**request object**" if passing data between different JSP pages.
- JSP declarations can be used to override **jspInit( )** and **jspDestroy( )** methods.

```
<%! public void  jspInit() {
      …….      }
    public void jspDestroy(){
    ……..      }                %>
```



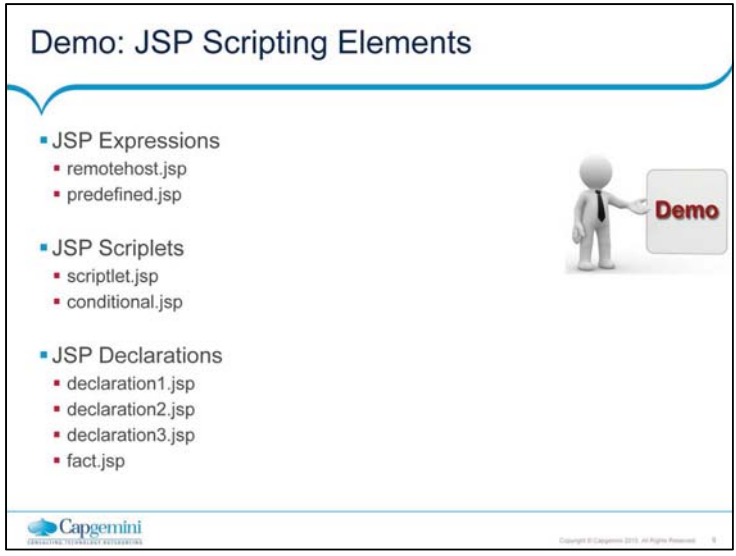
Predefined Variables:
To simplify code in **JSP expressions** and **scriptlets**, JSPs are supplied with eight automatically defined variables, sometimes are called **implicit objects**. The available variables are as follows:

- **request:** This is the **HttpServletRequest** associated with the request, and lets to look at the request parameters (via getParameter), the request type (GET, POST, HEAD, and so on), and the incoming HTTP headers (cookies, Referrer, and so on).
- **response:** This is the **HttpServletResponse** associated with the response to the client. Note that, since the output stream is buffered, it is legal to set HTTP status codes and response headers, even though this is not permitted in regular Servlets once any output has been sent to the client.
- **out:** This is the **PrintWriter** used to send output to the client. However, in order to make the response object useful, this is a buffered version of PrintWriter called **JspWriter**. Note that we can adjust the buffer size, or even turn buffering off, through use of the buffer attribute of the page directive. In scriptlets, need to refer to out explicitly in case we need to display anything.
- **session:** This is the **HttpSession object** associated with the session. As sessions are created automatically, this variable is bound even if there is no incoming session reference. The one exception is that if the session attribute of the page directive is used to turn sessions off. In this case, attempts to reference the session variable cause errors at the time the JSP page is translated into a servlet.



Predefined Variables:

- **Application:** This is the **ServletContext** as obtained via the **getServletConfig().getContext()**.
- **config:** This is the **ServletConfig** object for this page.
- **pageContext:** JSP introduced a new class called **PageContext** to encapsulate use of server-specific features like higher performance JspWriters. The advantage is that these can be accessed through this class rather than directly, code will still run on "regular" servlet/JSP engines.
- **page:** This is simply a synonym for **this**, and is not very useful in Java. It was created as a placeholder for the time when the scripting language could be something other than Java.
- **exception:** This implicit object applies only to JSP error pages – these are pages to which processing is forwarded when an exception is thrown from another JSP page. They must have the page directive **isErrorPage** attribute set to **true**. The implicit exception object is a **java.lang.Exception** instance that represents the uncaught exception that was thrown from another JSP page and that resulted in the current error page being invoked. The exception object is accessible only from the JSP error page instance to which processing was forwarded when the exception was encountered.

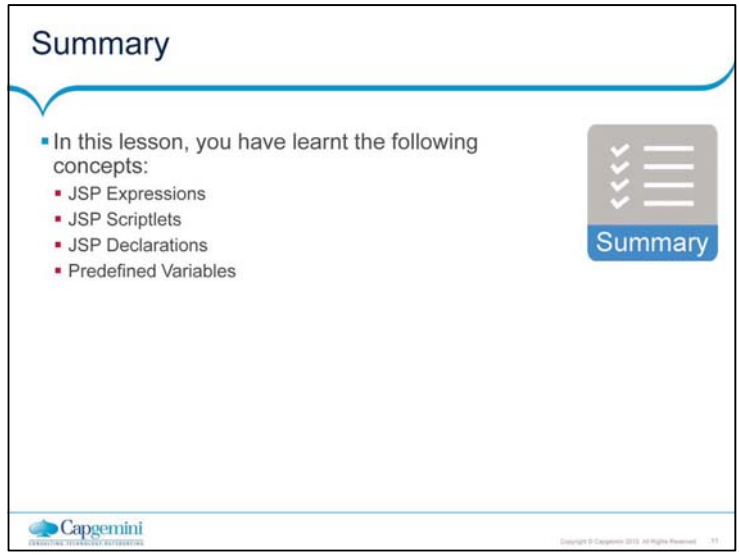


Deploy web application **Lesson3-JSPScriptlets** and show demo by executing each of the above JSP pages.

Lab: JSP Scripting Elements

- Lab 1.1
- Lab 1.2

Capgemini


Summary
In this lesson, you have learnt the following concepts:
JSP Expressions
JSP Scriptlets
JSP Declarations
Predefined Variables
Summary
Capgemini

Review – Questions

- Question 1: ___ inserts java code into _jspService( ) method.
- Question 2: A JSP expression can end with a semicolon (:) . True/False
- Question 3: ___ can be used to override jspInit( ) method.

Capgemini

Review – Questions

- Question 4: ___ is a buffered PrintWriter used in JSP to send output to client.
- Question 5: ___ implicit object is used to share data across users of the same web application.

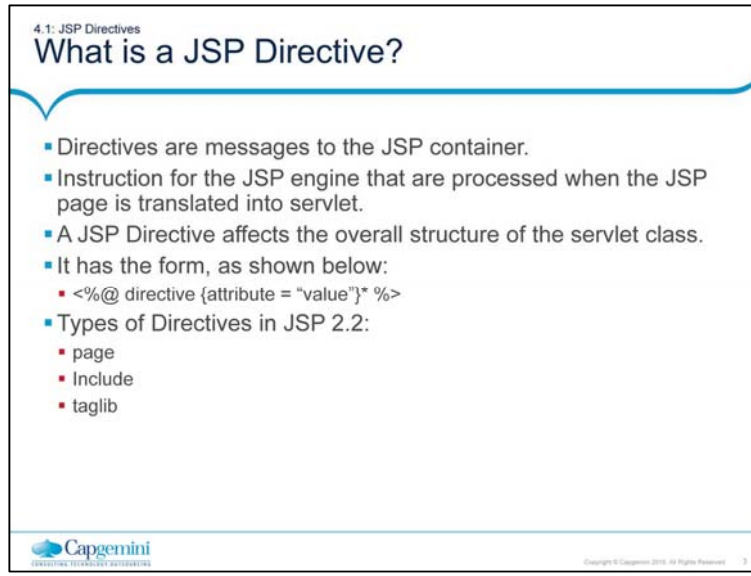
Java Server Pages (JSP)

Lesson 04: JSP Directives

Lesson Objectives

- In this lesson, you will learn about:
  - JSP Directives
    - What is a JSP Directive?
  - JSP Page Directive
  - JSP Include Directive

Capgemini

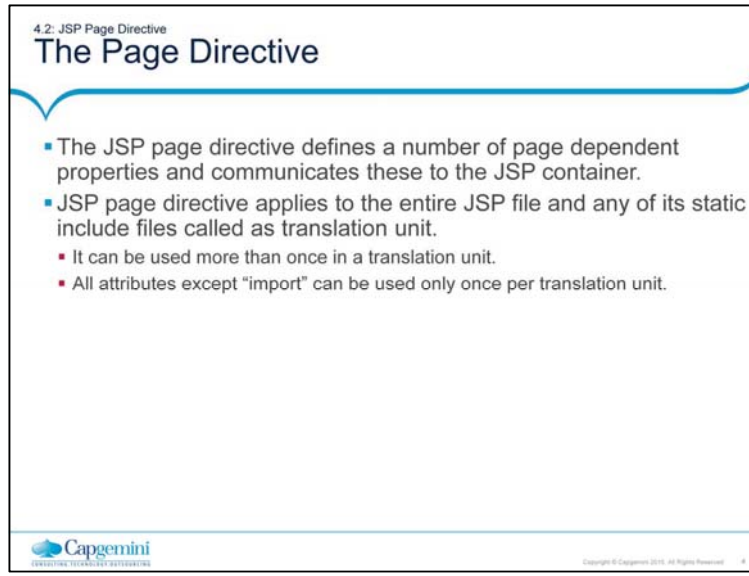


JSP Directives:

What is a JSP Directive?

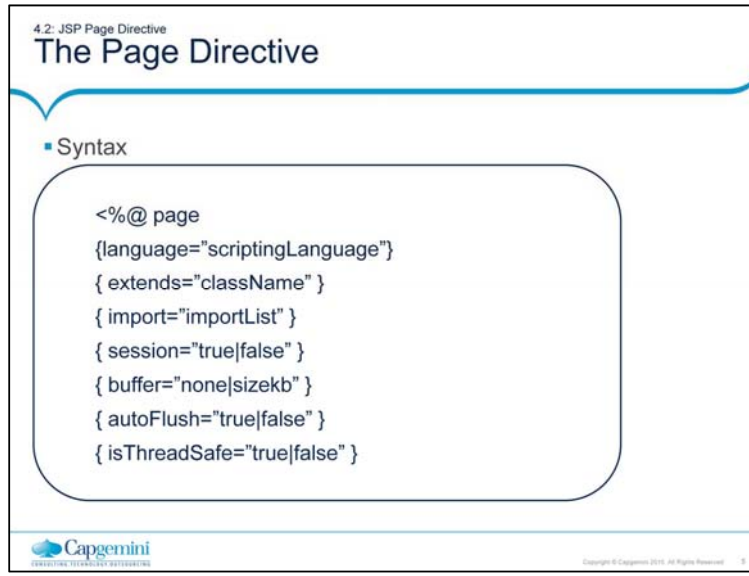
- A JSP *directive* affects the overall structure of the servlet class.
- Used to set global values such as class declarations methods to be implemented, output content type etc
- Do not produce any output to the client
- They have the scope of the entire file

- It usually has the following form: <%@ directive {attribute="value"}* %>
- There are three types of directives:
  - **page:** It allows to do things like import classes, customize the servlet super class, and the like; and
  - **include:** It allows to insert a file into the servlet class at the time the JSP file is translated into a servlet.
  - **taglib:** It is intended to let JSP authors define their own tags. This directive is explained in detail in the lesson on "Custom Tags in JSP Pages".



JSP Page Directive:

- The **<%@ page %>** directive applies to an entire **JSP file** and any of its **static include files**, which together are called a **translation unit**. A static include file is a file whose content becomes part of the calling JSP file. The <%@ page %> directive does not apply to any dynamic include files.
- The **<%@ page %>** directive can be used more than once in a translation unit. However, each attribute, except import, can be used once. Since in Java programming language, the **import attribute** is similar to the **import statement**, a <%@ page %> directive with import more than once can be used in a JSP file or translation unit.
- No matter the position of <%@ page %> directive in a JSP file or included files, it applies to the entire translation unit. However, it is often good programming style to place it at the top of the JSP file.

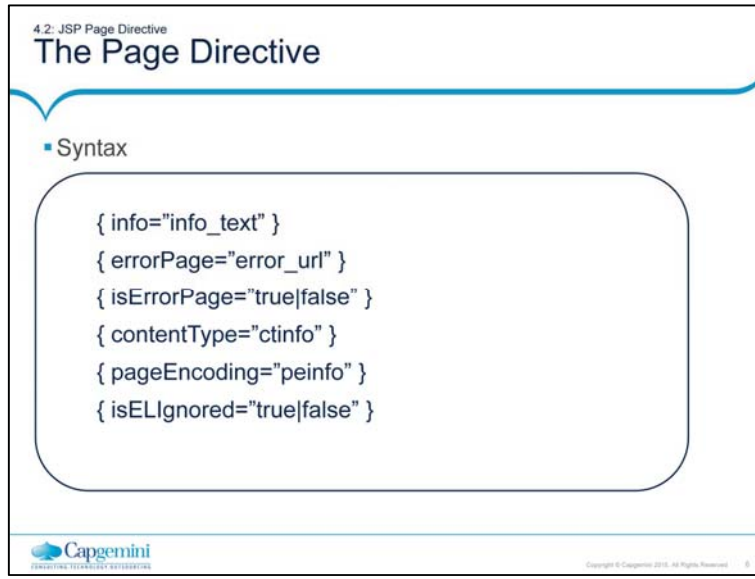


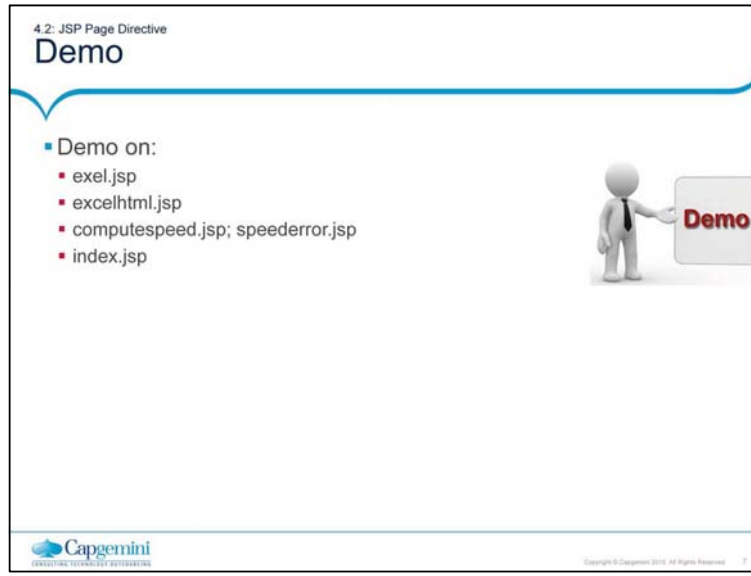
JSP Page Directive:
Attributes of JSP Page directive:

- *language*=**"java" :** It indicates the scripting language used in scriptlets, declarations, and expressions in the JSP file and any included files. In this release, the only allowed value is java.
- *extends*=**"package.*class*" :** It indicates the fully qualified name of the superclass of the Java class file this JSP file will be compiled to. Use this attribute cautiously, as it can limit the JSP container's ability to provide a specialized superclass that improves the quality of the compiled file.
- *import*=**"{package.class | package.*}, ..."** : It indicates a comma-separated list of Java packages that the JSP file should import. The packages (and their classes) are available to scriptlets, expressions, and declarations within the JSP file. If more than one package to be imported, then specify a comma-separated list after import or use import more than once in a JSP file. The following packages are implicitly imported, so it does not need to specify them with the import attribute: java.lang.* , javax.servlet.*, javax.servlet.jsp.*, javax.servlet.http.*
  Place the import attribute before the element that calls the imported class.
  For example:

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

The import attribute is the only one that is allowed to appear multiple times.


4.2: JSP Page Directive
The Page Directive
Syntax
{ info="info_text" }
{ errorPage="error_url" }
{ isErrorPage="true|false" }
{ contentType="ctinfo" }
{ pageEncoding="peinfo" }
{ isELIgnored="true|false" }
Capgemini

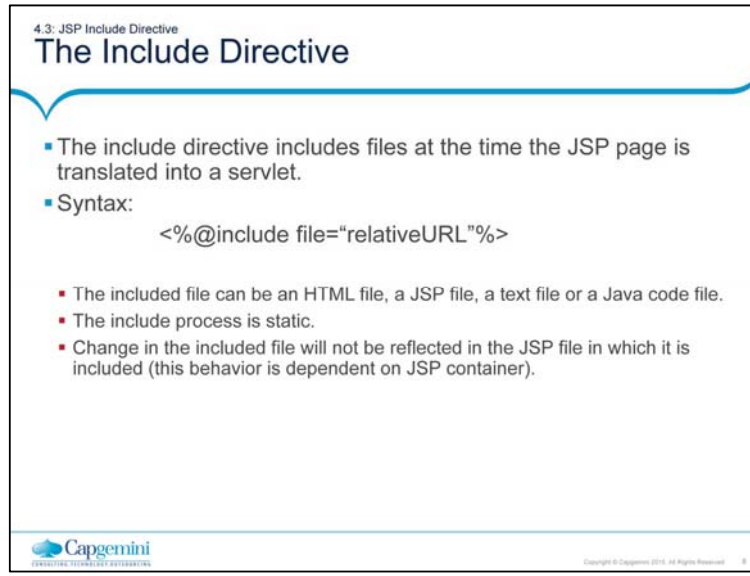


Deploy web application **Lesson4-JSPDirectives** and show demo by executing each of the above JSP pages.

Content-Type Demo : exel.jsp & Excelhtml.jsp will display the content in excel format

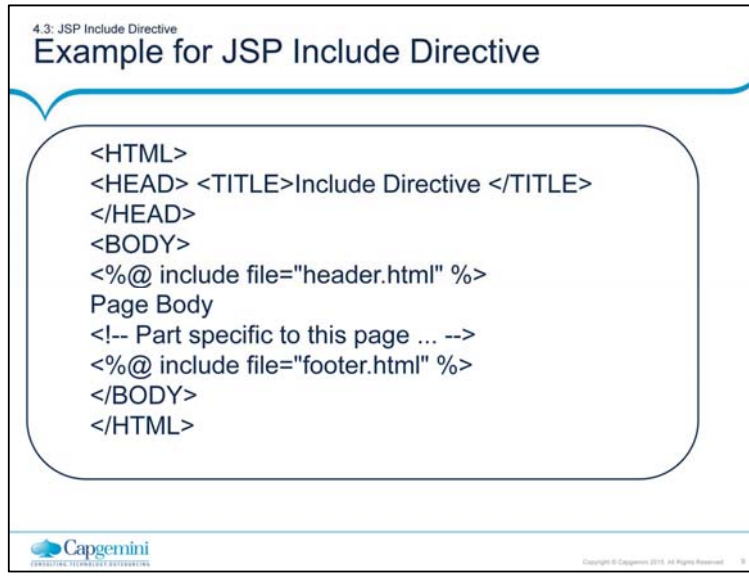
Usage of Error page Demo: Execute computspeed.jsp page, if submit the form without any value or with invalid values then the NumberFormatException will be thrown which will be passed on to the error page (speederror.jsp) as it is set as an error page in the “isErrorPage” attribute of the page directive. If the valid values are entered in the textboxes speed will be displayed in the output.

Explain the error page configuration in the web.xml as given in the Demo.



The JSP Include Directive:

- The **<%@ include %>** directive inserts a file of text or code in a JSP file at translation time, when the JSP file is compiled. When the **<%@ include %>** directive is used, the include process is **static**.
  - A static include means that the text of the included file is added to the JSP file. The included file can be a JSP file, HTML file, or text file. If the included file is a JSP file, then its JSP elements are parsed and their results are included (along with any other text) in the JSP file.
- We can only use **include** to include static files.
  - This means that the parsed result of the included file is added to the JSP file where the <%@ include %> directive is placed.
  - Once the included file is parsed and included, processing resumes with the next line of the calling JSP file.
- A JSP container can include a mechanism for being notified if an included file changes, so the container can recompile the JSP page. However, the JSP 2.1 specification does not have a way of directing the JSP container that included files have changed.
- The included file must not contain <html>, </html>, <body>, or </body> tags. Since the entire content of the included file is added at that location in the JSP file, these tags would conflict with the same tags in the calling JSP file, causing an error.



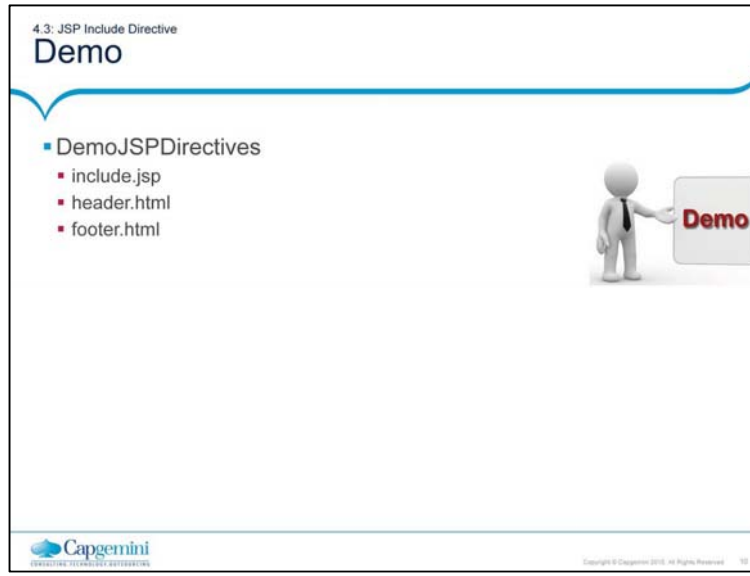
The JSP Include Directive:
Syntax for JSP Include Directive:

```
<%@ include file="relativeURL" %> file="relativeURL"
```

- The pathname to the included file is always a relative URL. A relative URL is just the path segment of an URL, without a protocol, port, or domain name. An example is shown below:

```
"error.jsp" , "/templates/onlinestore.html" , "/beans/calendar.jsp"
```

- If the relative URL starts with /, then the path is relative to the JSP application's context, which is a **javax.servlet.ServletContext** object that is in turn stored in the application object. If the relative URL starts with a directory or file name, then the path is relative to the JSP file.
- The example on the slide shows how we can add header and footer to our **include.jsp** page by making use of the **include** directive to include **header.html** and **footer.html**.
- Note that the include directive inserts the files at the time the page is translated. Hence, if the header and footer changes, then it needs to re-translate all the JSP pages that refer to it. If, however, the included files changed more often, then the **jsp:include** action should be used instead. This action includes the file at the time the JSP page is requested, and is discussed in the next lesson.



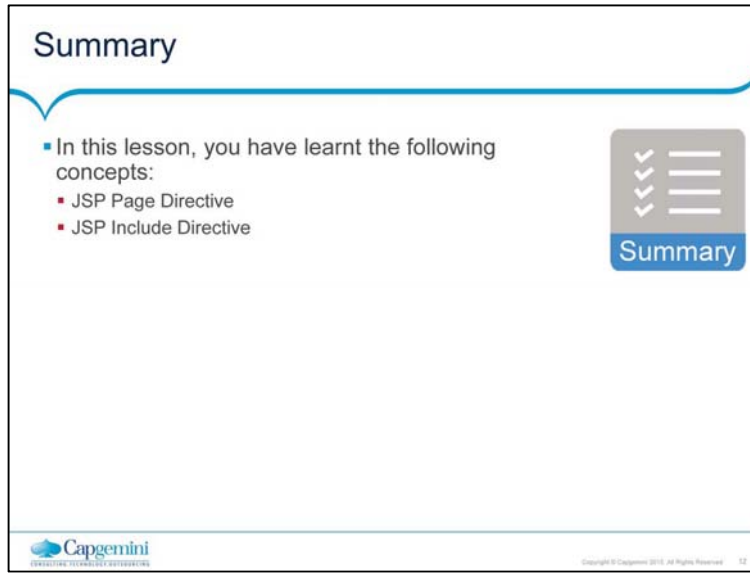
Deploy web application **Lesson4-JSPDirectives** and show demo by executing each of the above JSP pages.

The include.jsp page includes the header.html and the footer.html pages using the include directive (@ include). Invoke include.jsp page to run the demo.

Lab

- Lab 1.3

Capgemini


Summary
In this lesson, you have learnt the following concepts:
JSP Page Directive
JSP Include Directive
Summary
Capgemini

Review Questions

- Question 1: Directives are ___ to JSP Container.
- Question 2: Which directive helps you to import classes?
- Question 3: If you want the JSP container to send client request one at a time to the JSP page, then which attribute of the page directive will need to be set?

Capgemini

Review Questions

- Question 4: The default value for the buffer attribute of the page directive is ___ kb.
  Question 5: JSP include directive follows a ___ include process:
  - Option 1: static
  - Option 2: dynamic
  - Option 3: both

Capgemini

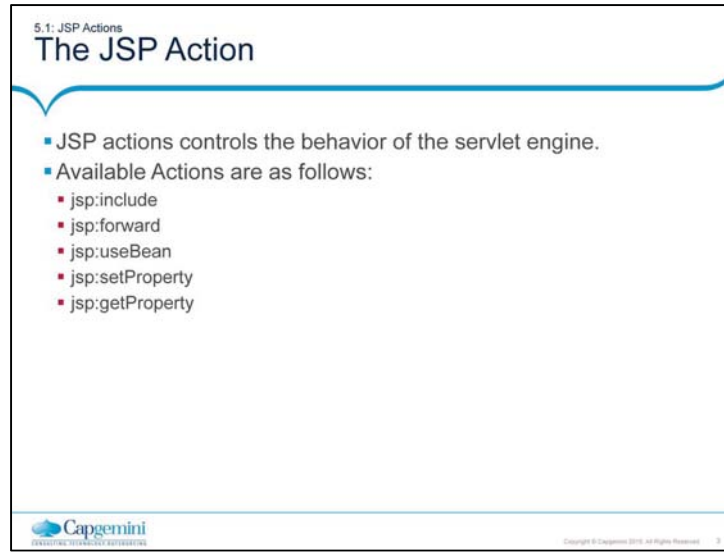
Java Server Pages (JSP)

Lesson 05 : JSP Actions

Lesson Objectives

- In this lesson, you will learn the following concepts:
  - JSP Actions
  - jsp:include Action
    - include Action Vs Directive
  - jsp:forward Action
    - Integrating Servlets & JSP
    - jsp:forward Vs response.sendRedirect
  - Java Bean
  - Bean Related JSP Actions

Capgemini

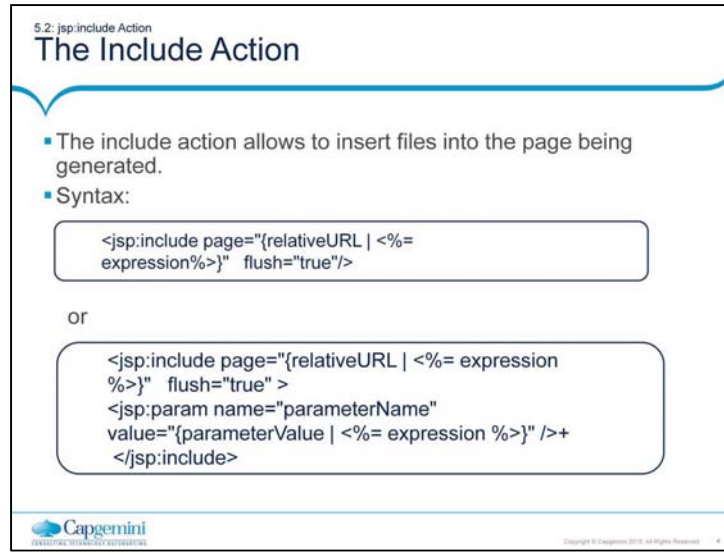


JSP Actions:

What is a JSP Action?

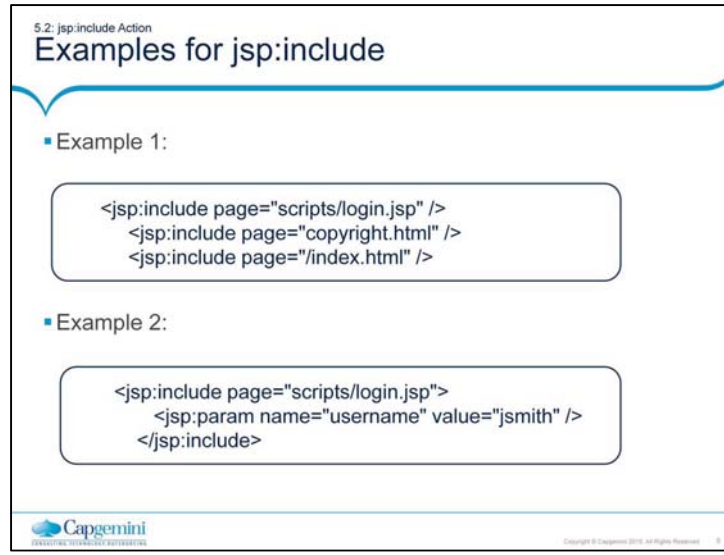
- **JSP actions** use constructs in XML syntax to control the behavior of the servlet engine. With JSP actions, following actions can be performed:
  - Dynamically insert a file
  - Reuse JavaBeans components
  - Forward the user to another page
  - Generate HTML for the Java plugin
- Available actions include:
  - **jsp:include** : It includes a file at the time the page is requested.
  - **jsp:forward :** It forwards a client request to an HTML file, JSP file, or servlet for processing.
  - **jsp:useBean** : It finds or instantiates a JavaBean.
  - **jsp:setProperty** : It sets the property of a JavaBean.
  - **jsp:getProperty** : It inserts the property of a JavaBean into the output.

- The above actions are described in more detail further in this lesson.



The jsp:include Action:

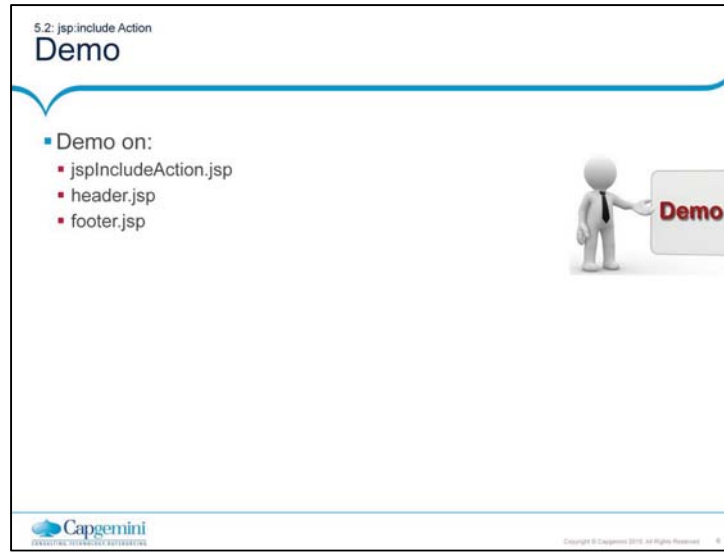
- The **<jsp:include>** element allows to include either a **static** or **dynamic** file in a JSP file. The results of including static and dynamic files are quite different.
  - If the file is static, its content is included in the calling JSP file.
  - If the file is dynamic, it acts on a request and sends back a result that is included in the JSP page.
- When the include action is finished, the JSP container continues processing the remainder of the JSP file.
- We cannot always determine from a pathname if a file is static or dynamic. For example: http://server:8080/index.html might map to a dynamic servlet through a Web server alias. The **<jsp:include>** element handles both types of files, so it is convenient to use when we do not know whether the file is static or dynamic.
- If the included file is dynamic, we can use a **<jsp:param>** clause to pass the name and value of a parameter to the dynamic file. As an example, we can pass the string username and a user's name to a login form that is coded in a JSP file.
- The above slide lists the syntax of **jsp:include** action. The value for the page attribute is a **relative URL** that locates the file to be included, or an **expression** that evaluates to a String equivalent to the relative URL. The relative URL looks like a pathname - it cannot contain a protocol name, port number, or domain name. The URL can be absolute or relative to the current JSP file. If it is absolute (beginning with a /), then the pathname is resolved by Web or application server.
- We must include the attribute **flush="true"**, as it is not a default value. We cannot use a value of false. Use the **flush** attribute exactly as shown in the slide.



The jsp:include Action:

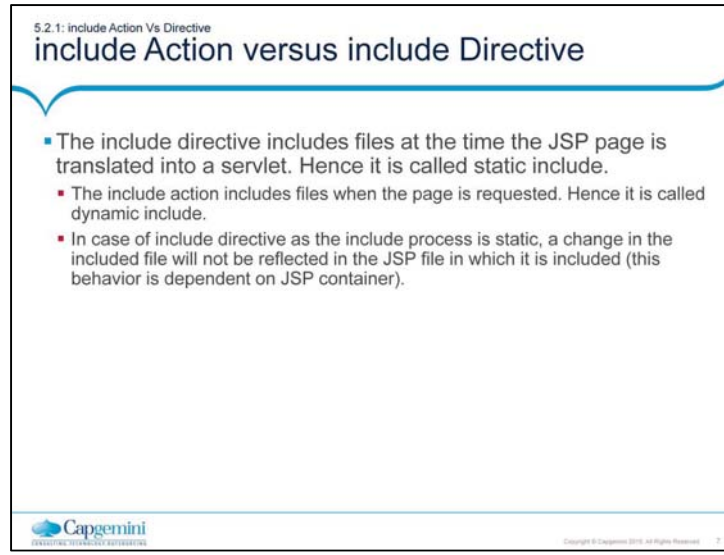
Examples for jsp:include:

- The above slide lists some examples of **jsp:include** action with the page attribute value as a relative URL.
- In the second example on the slide, the **<jsp:param>** clause is used which allows to pass one or more name / value pairs as parameters to an included file. The included file should be dynamic, that is a JSP file, servlet, or other file that can process the parameter.
- We can use more than one **<jsp:param>** clause if we want to send more than one parameter to the included file. The name attribute specifies the parameter name and takes a case-sensitive literal string. The value attribute specifies the parameter value and takes either a case-sensitive literal string or an expression that is evaluated at request time.



Deploy web application **Lesson5-JSPActions** and show demo by executing each of the above JSP pages.

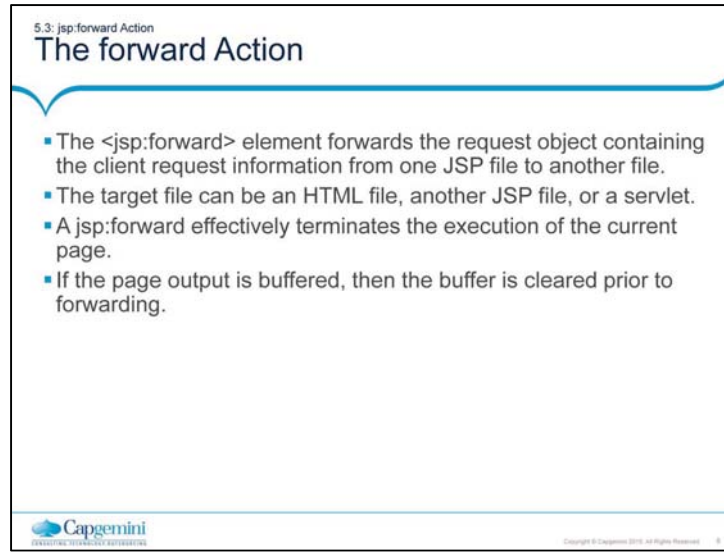
Note: Here is an example, where a JSP page uses jsp:include action to inserts header and footer into jspIncludeAction.jsp



The jsp:include Action:

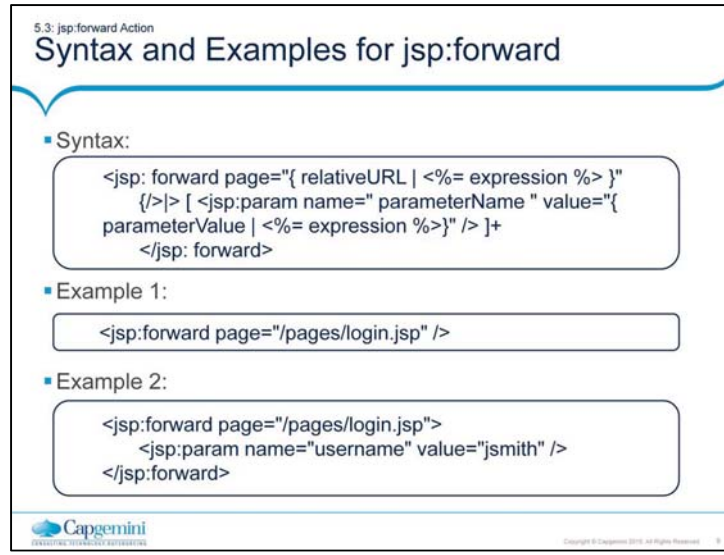
The JSP Include Action versus JSP Include Directive:

- Unlike the **include directive**, which inserts the file at the time the JSP page is "translated" into a servlet, the **include action** inserts the file at the time the page is "requested". This pays a small penalty in efficiency, and precludes the included page from containing general JSP code (for example, it cannot set HTTP headers), but it gains significantly in flexibility.
- A JSP container can include a mechanism for being notified if an included file changes (in case of include directive), so the container can recompile the JSP page. However, the JSP 2.2 specification does not have a way of directing the JSP container that included files that have changed.
- In the recent versions of the **Server containers** (such as JBOSS, Tomcat, Websphere Application server), this mechanism has been implemented. Hence there is no change in the behavior of **include directive** and **include action**. Both include the files dynamically, hence change in the included file is reflected in the original page in both the cases. However, it is recommended that we should not rely on the container's behavior and always use **include action** whenever a **dynamic include** is required.



The jsp:forward Action:

- The **<jsp:forward>** element forwards the **request object** containing the client request information from one JSP file to another file. The target file can be an HTML file, another JSP file, or a servlet, as long as it is in the same application context as the forwarding JSP file. The lines in the source JSP file after the **<jsp:forward>** element are not processed.
- We can pass parameter names and values to the target file by using a **<jsp:param>** clause. An example of this would be passing the parameter name ***username*** (with name="username") and the value ***scott*** (with value="scott") to a jsp login file as part of the request. If we use **<jsp:param>**, then the target file should be a dynamic file that can handle the parameters.
- Be careful while using **<jsp:forward>** with unbuffered output. If the **<%@ page %>** directive has used with **buffer=none** to specify that the output of the JSP file should not be buffered, and if the JSP file has any data in the out object, using **<jsp:forward>** will cause an IllegalStateException.



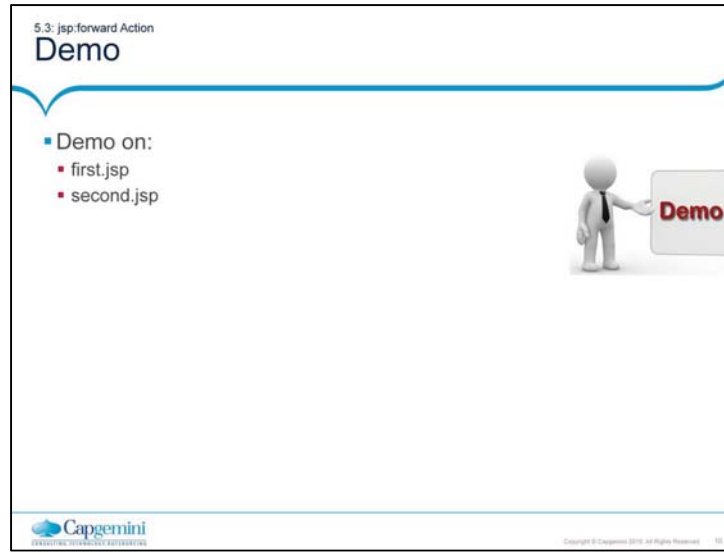
The jsp:forward Action:

The above slide lists the syntax for **jsp:forward** action:

- Like **jsp:include**, the value for the page attribute in case of jsp:forward is a **relative URL** or an **expression** representing the file to which we are forwarding the request. The file can be another JSP file, a servlet, or any other dynamic file that can handle a request object.
- The **jsp:param** element is similar to the one that we saw for jsp:include action.

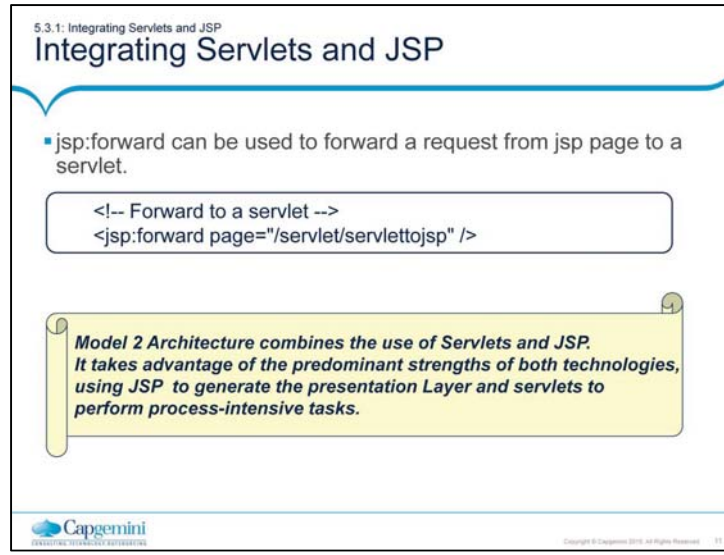
Examples:

- The above slide shows examples of **jsp:forward** action.
  - The first example depicts how a request can be forwarded to another jsp page.
  - In the second example, the request is forwarded to another jsp and a request parameter "username" is also passed to the forwarded page. In the **login.jsp** page, the username request parameter can be obtained by using the following method:
    - request .getParameter ("username")



Deploy web application **Lesson5-JSPActions a**nd show demo by executing each of the above JSP pages in **forward** folder.

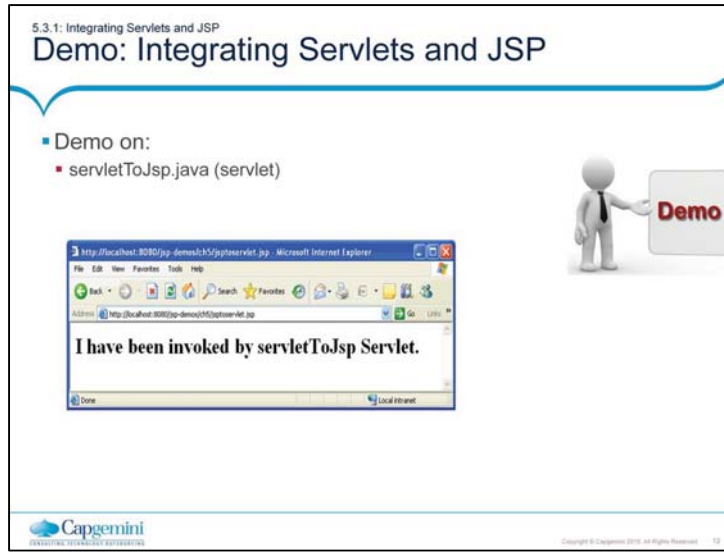
```
<html>
 <body bgcolor="white">
 <font color="red"> VM Memory usage > 50%.
 </html>
```



The jsp:forward Action:

Integrating Servlets and JSP:

- The Model 2 architecture, discussed in lesson 1, is a hybrid approach for serving dynamic content, since it combines the use of both servlets and JSP. It takes advantage of the predominant strengths of both technologies, using JSP to generate the presentation layer and servlets to perform **process-intensive tasks**. On a more primitive level, we have seen how to integrate JSPs and servlets together using the **jsp:forward** tag.
- **Example:** A request to **jsptoservlet.jsp** forwards the request to the **servletToJsp** servlet which in turn sets the attribute in the request to contain the servlet name and obtains the **RequestDispatcher** to **hello.jsp** page and forwards the request using **RequestDispatcher**. Upon receiving the request, the **hello.jsp** shows the servlet name.
- The above slide shows the code fragment from the **jsptoservlet.jsp** file. The next slide shows the code listing for the **servlet** and **hello.jsp** files.

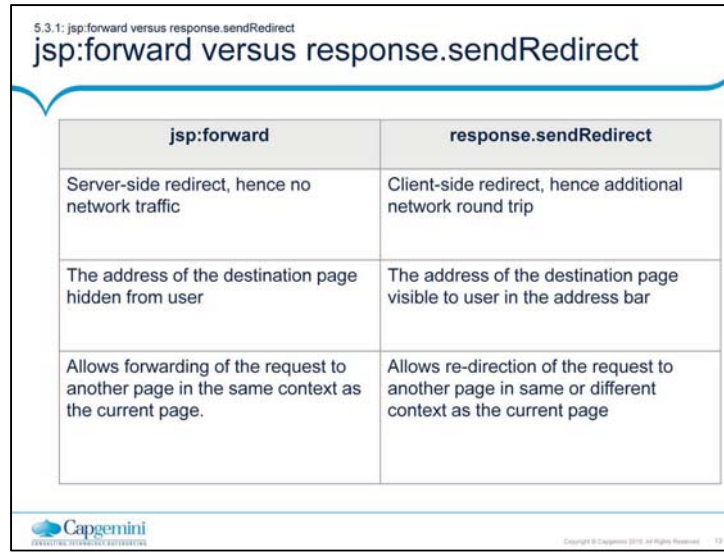


Note: Following is the code fragment from servletToJsp servlet which forwards the request and response to hello.jsp page.

```
public void doGet (HttpServletRequest request,
                   HttpServletResponse response) {
try {
        // Set the attribute and Forward to hello.jsp
      request.setAttribute ("servletName", "servletToJsp");
      getServletConfig().getServletContext().getRequestDispatcher(
                "/pages/hello.jsp").forward(request, response);
     } catch (Exception ex) {
                ex.printStackTrace ();
     } }
```

The following example shows a code fragment from **hello.jsp**, which displays the servlet name passed by **servletToJsp** servlet.

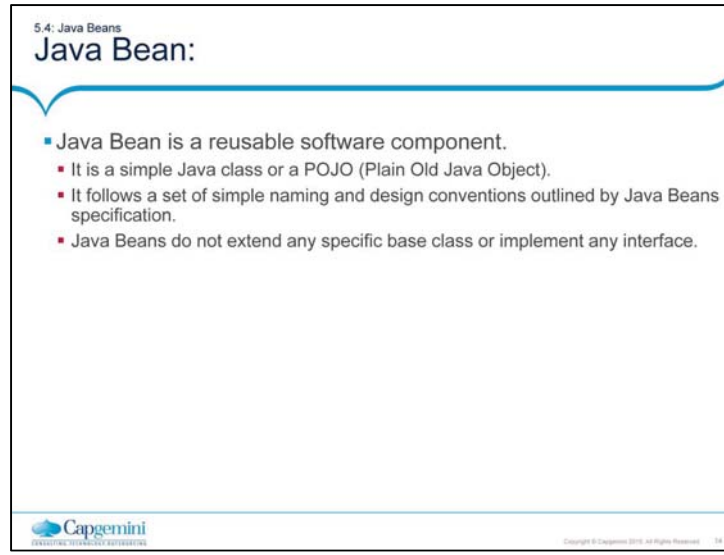
```
<body bgcolor="white">
<h1> I have been invoked by
<% out.print (request.getAttribute("servletName").toString()); %>
Servlet. </h1>
```



The jsp:forward Action:

jsp:forward versus response.sendRedirect:

- **forward** is server side redirect and **sendRedirect** is client side redirect. When we invoke a forward request, the request is sent to another resource on the server, without the client being informed that a different resource is going to process the request. This process occurs completely within the web container, and then returns to the calling method. When a **sendRedirect** method is invoked, it causes the web container to return to the browser indicating that a new URL should be requested. Since the browser issues a completely new request, any objects that are stored as **request attributes** before the redirect occurs will be lost. As a result of this extra round trip, a redirect is slower than forward. Client can disable sendRedirect.
- In case of **sendRedirect**, the user sees only the address of the destination page and can bookmark it and access it independently.
- In case of forward, the request can be forwarded to another page in the same context/domain as the current page whereas in case of **sendRedirect** we can redirect the request to a page in another context or domain as well. for e.g. if **sendRedirect** was called at www.mydomain.com then it can also be used to redirect a call to a resource on www.yourdomain.com.



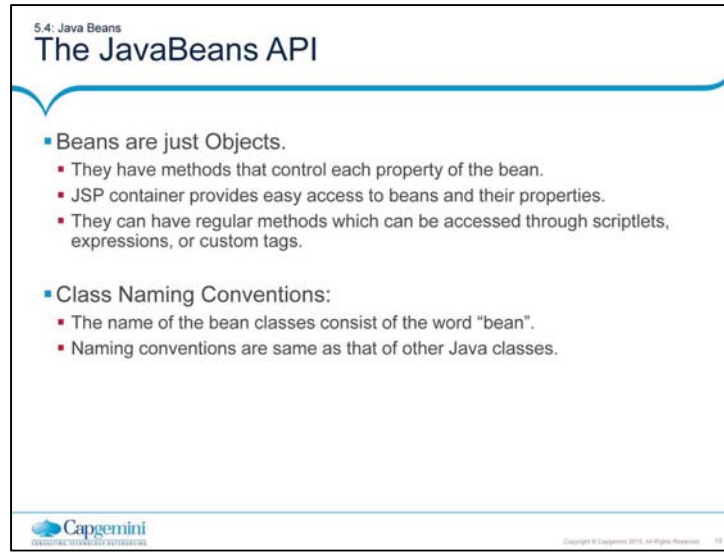
Java Beans:

Building own JSP components:

- The **JavaServer Pages (JSP)** component model is centered on Java software components called **Beans**, which must adhere to specifications outlined in the **JavaBeans API**. The JavaBeans API, created by Sun Microsystems with industry cooperation, dictates the rules that software developers must follow to create stand-alone, reusable Java software components. By using JSP's collection of Bean tags, content developers can use the power of Java to add dynamic elements to their pages without writing a single line of code.

So, what makes a Bean so special?

- A Bean is simply a **Java class** or a **POJO** that follows a set of simple naming and design conventions outlined by the JavaBeans specification. Beans are not required to extend a specific base class or implement a particular interface. If a class follows the Bean conventions, and we treat it like a Bean, then it is a Bean. A particular, good feature of the Bean conventions is that they are rooted in sound programming practices. In the next section, we will discuss these conventions and show how to create Beans.



Java Beans:

The JavaBeans API:

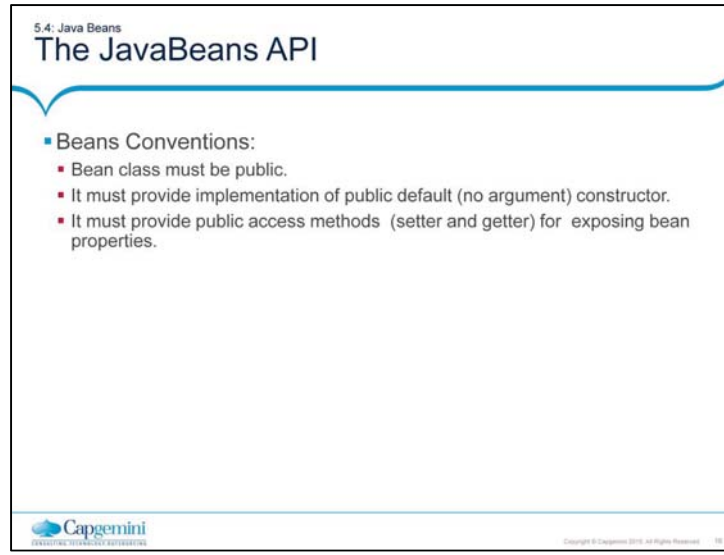
- If we follow the conventions specified by the **JavaBeans API**, then the JSP container can interact with **Beans** at a programmatic level, even though the container has no real understanding of what the Bean does or how it works.

Bean Are Just Objects:

- As with any other Java class, instances of Bean classes are simply Java objects. Hence, we can reference Beans and their methods directly through Java code in other classes or through JSP scripting elements. Since they follow the Bean conventions, we can work with Beans without having to write Java code. Bean containers, such as a JSP container, can provide easy access to Beans and their properties. Following the JavaBeans API coding conventions, means creating methods that control access to each property we wish to define for our Bean. Beans can also have regular methods like any other Java object which can be accessed through scriptlets, expressions, or custom tags.

Class Naming Conventions:

- Note that in most examples Bean classes often include the word Bean in their name, such as UserBean, ClockBean, DataAccessBean. This is not a rule hence not mandatory. Beans follow the same class naming rules as other Java classes. They must start with an alphabetic character, contain only alphanumeric and underscore characters, and be case-sensitive. Additionally, like other Java classes, it is common (but not required) to start the name of a Bean class with a capital letter.



Java Beans:

Bean Conventions:

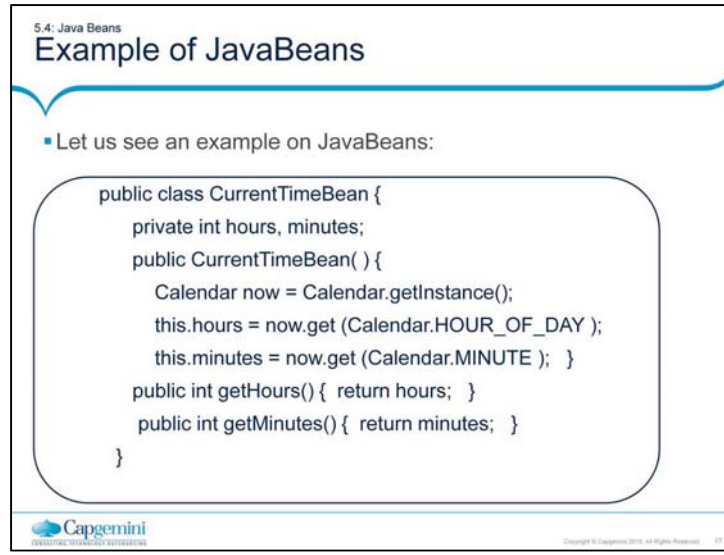
- The **Bean conventions** enable us to develop Beans, because they allow a Bean container to analyze a Java class file and interpret its methods as properties, designating the class as a Bean. The conventions dictate rules for defining a Bean's constructor and the methods that will define its properties.

The Bean Constructor:

- The first rule of JSP Bean building is that we must implement a constructor that takes no arguments. This constructor is used by the JSP container to instantiate the Bean through the **<jsp:useBean>** tag. If a class does not explicitly specify any constructors, then a default zero-argument constructor is assumed. As a result of this default constructor rule, the following Java class is perfectly valid and technically satisfies the Bean conventions:
  - *public class DoNothingBean { }*

The Magic of Introspection:

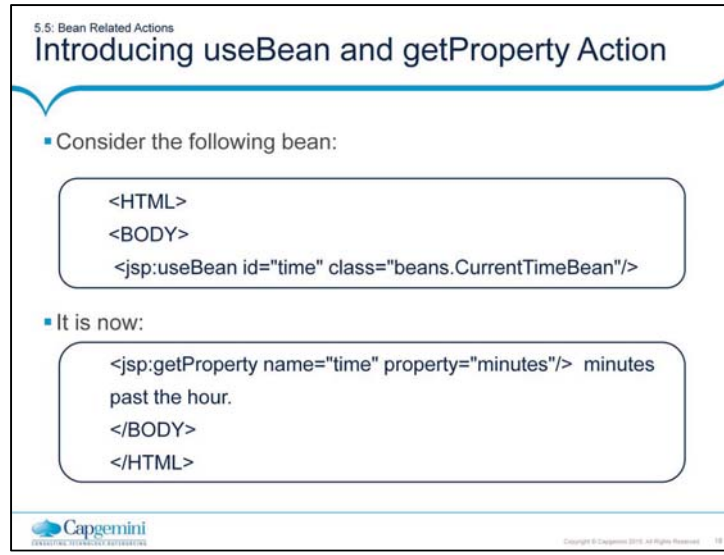
- The **Bean container** can interact with the bean through introspection. This introspection process happens at run-time and allows a class to expose its properties on request. One way in which introspection can occur is through "Reflection". The Bean container determines the properties that a Bean supports by analyzing its public methods for the presence of **property access methods** that meet criteria defined by the JavaBeans API.
- For a property to exist, its Bean class must define an **access method** to return the value of the property, change the value of the property, or both. It is the presence alone of the specially named **access methods** that determines a Bean class' properties.



Java Beans:

Example of Java Bean:

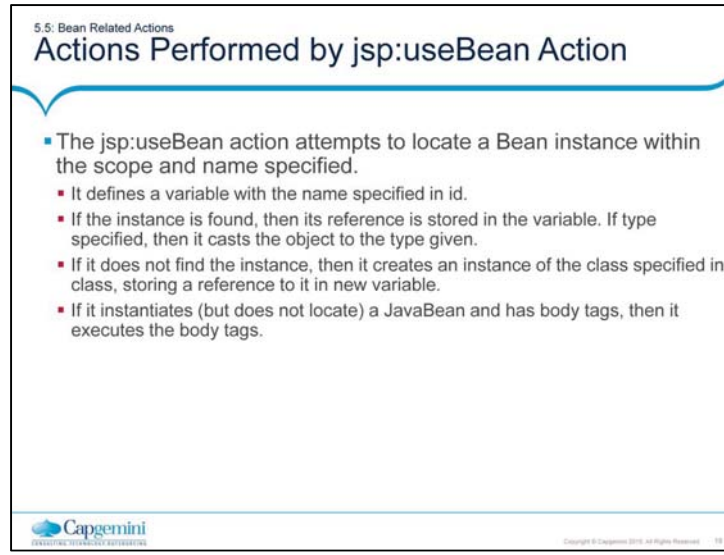
- The example on the slide shows how a **CurrentTimeBean** is coded. This bean follows all the conventions. It consists of a public **no-argument constructor** that initializes the value for the private instance variables hours and minutes. It has the public access methods for two properties, namely **hours** and **minutes**. Since it has only the **getter** methods for these properties, they will be exposed as read-only properties.
- The two methods simply return the appropriate values as stored in the instance variables. Since these methods meet the Bean conventions for naming access methods, we have just defined two properties that we can access through JSP Bean tags.
- Properties should not be confused with instance variables, even though instance variables are often mapped directly to property names. Properties of a Bean are not required to correspond directly with instance variables. A Bean's properties are defined by the method names themselves, not the variables or implementation behind them. This leaves the Bean designer free to alter the inner workings of the Bean without altering the interface and collection of properties exposed to users of the Bean.



Bean Related Actions:

Accessing the Java Bean and its property in the JSP page:

- The code shown in the above slide depicts how we can use the **jsp:useBean** action to create an instance of the **CurrentTimeBean** Java bean.
- It uses **jsp:getProperty** action to access the minutes properties. Internally the bean container will invoke the **getMinutes()** method of the **CurrentTimeBean** to retrieve the property value as a string.



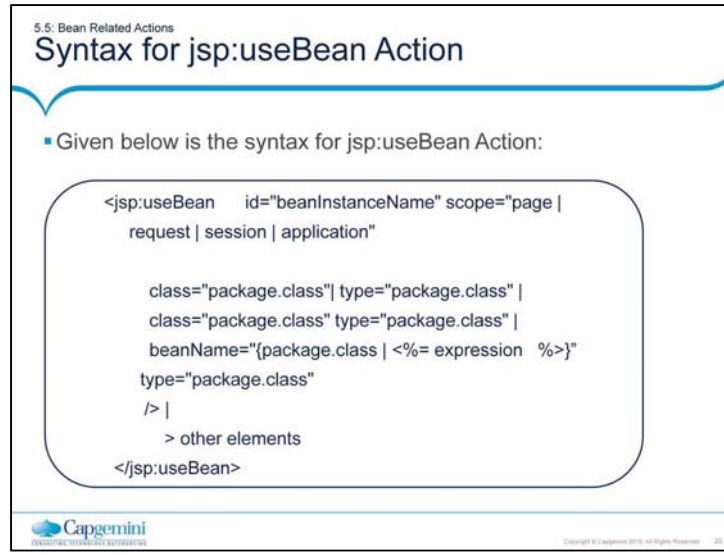
Bean Related Actions:

The <jsp:useBean> Action

- A **<jsp:useBean>** action associates an instance of a Java object defined within a given scope available with a given id via a newly declared scripting variable of the same id.
- The action tries to find an existing object using id and scope. If it is not found, then it will attempt to create the object using the other attributes.

The actions performed are:

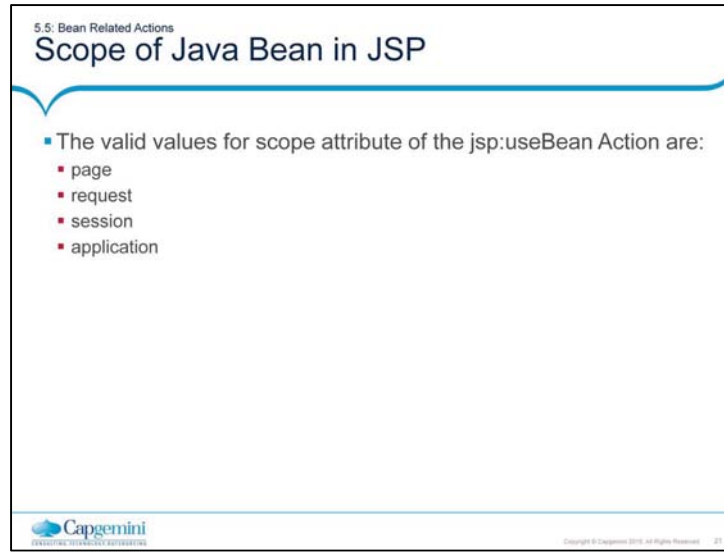
- It attempts to locate an object based on the attribute values (id, scope).
- It defines a scripting language variable with the given id in the current **lexical scope** of the scripting language of the specified type (if given) or class (if type is not given).
- If the object is found, then the variable's value is initialized with a reference to the located object, cast to the specified type. If the cast fails, then a **java.lang.ClassCastException** occurs. This completes the processing of this **useBean** action.
- If the **jsp:useBean** element has a non-empty body, then it is ignored. This completes the processing of this **useBean** action.
- If the object is not found in the specified scope and neither class nor beanName are given, then a **java.lang.InstantiationException** occurs. This completes the processing of this **useBean** action.



Bean Related Actions:

The Jsp:useBean Action - Attributes and Usage:

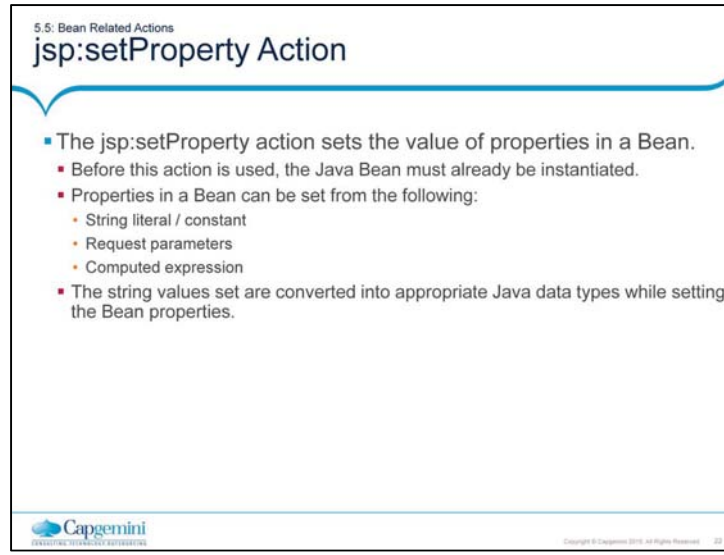
- **id="beanInstanceName"** : It is a variable that identifies the Bean in the scope specified. The variable name can be used in expressions or scriptlets in the JSP file. The name is case sensitive and must conform to the naming conventions of the scripting language used in the JSP page. If the Bean has already been created by another <jsp:useBean> element, then the value of id must match the value of id used in the original <jsp:useBean> element.
- **class="package.class":** It instantiates a Bean from a class, using the "**new**" keyword and the **class constructor**. The class must not be abstract and must have a public, no-argument constructor.
- **type="package.class"** : If the Bean already exists in the scope, then it gives the Bean a data type other than the class from which it was instantiated. If we use type without class or beanName, then no Bean is instantiated.
- **class="package.class" type="package.class"** : It instantiates a Bean from the class named in class, and assigns the Bean the data type specified in type. The value of type can be the same as class, a superclass of class, or an interface implemented by class.
- **beanName="{package.class | <%= expression %>}" type="package.class"** : It instantiates a Bean from either a class or a serialized template, using the **java.beans.Beans.instantiate** method, and gives the Bean the type specified in type.
- **scope**: This attribute is explained in the next slide.



Bean Related Actions:

The Jsp:useBean Action - Attributes and Usage:

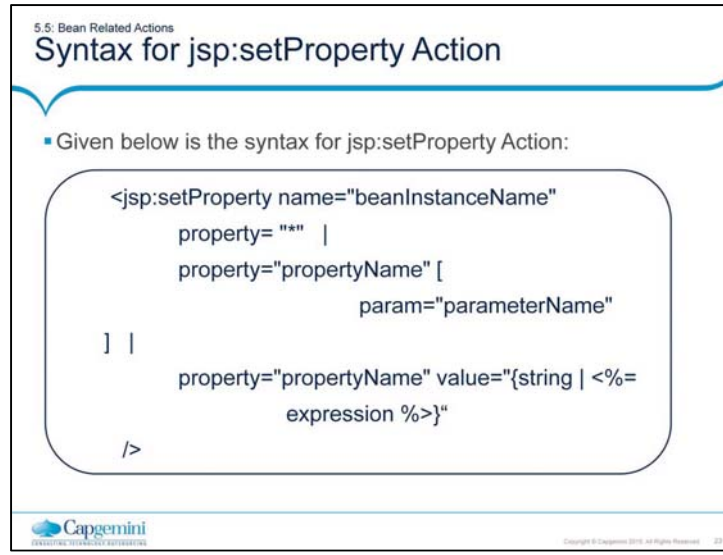
- scope="**page** | request | session | application" : It indicates the scope in which the Bean exists and the variable named in id is available. The default value is page. The meanings of the different scopes are as follows:
  - **page:** In this scope the Bean can be used within the JSP page with the **<jsp:useBean>** element or any of the page's static include files, until the page sends a response back to the client or forwards a request to another file.
  - **request:** In this scope the Bean can be used from any JSP page processing the same request, until a JSP page sends a response to the client or forwards the request to another file. We can use the request object to access the Bean.
    For example: request.getAttribute(*beanInstanceName*)
  - **session:** In this scope the Bean can be used from any JSP page in the same session as the JSP page that created the Bean. The Bean exists across the entire session, and any page that participates in the session can use it. The page in which the Bean is created must have a **<%@ page %>** directive with **session=true**.
  - **application:** In this scope the Bean can be used from any JSP page in the same application as the JSP page that created the Bean. The Bean exists across an entire JSP application, and any page in the application can use the Bean.



Bean Related Actions:

The jsp:setProperty Action:

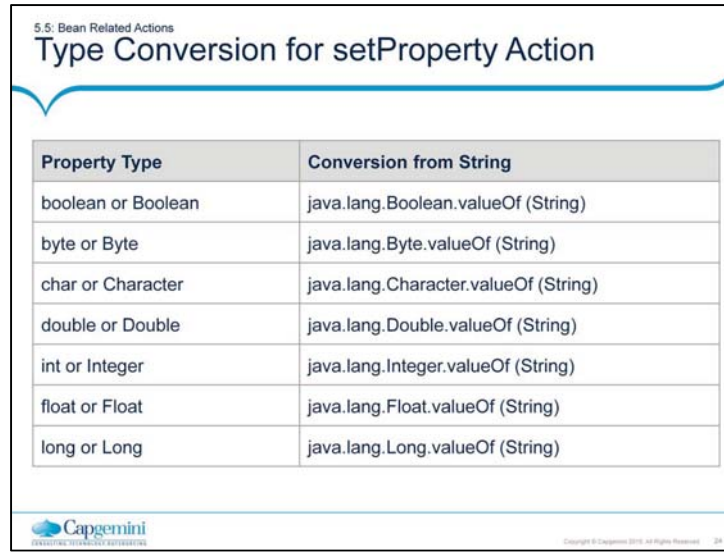
- The **jsp:setProperty action** sets the value of properties in a Bean. The **name** attribute denotes an object that must be defined before this action appears.
- There are two variants of the **jsp:setProperty action**. Both variants set the values of one or more properties in the Bean based on the type of the properties. The usual Bean introspection is done to discover what properties are present, and for each property, its name, whether they are simple or indexed, their type, and setter and getter methods are determined.
- Properties in a Bean can be set from either of the following:
  - one or more parameters in the request object
  - a String constant
  - a computed request-time expression
- Simple and indexed properties can be set using **setProperty**.
- The string values set are converted into appropriate Java data type while setting the Bean properties using its setter method. The conversion applied is listed in subsequent slide.
- While assigning values to **indexed properties**, the value must be an **array**.



Bean Related Actions:

The jsp:setProperty Action - Attributes and Usage

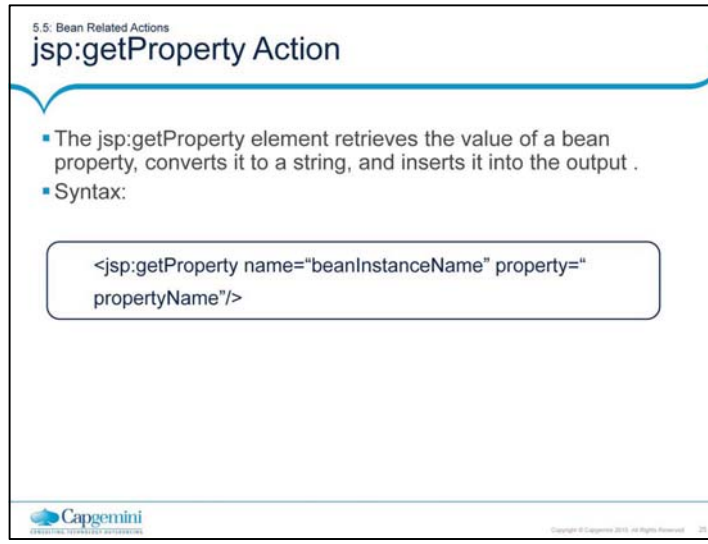
- **name="beanInstanceName"**: It indicates the name of an instance of a Bean that has already been created or located with a <jsp:useBean> element.
- **property="*"** : It stores all the values that the user enters in the viewable JSP page (called **request parameters**) in matching Bean properties. The names of the properties in the Bean must match the names of the **request parameters**, which are usually the elements of an HTML form. If a request parameter has an empty or null value, then the corresponding Bean property is not set. Likewise, if the Bean has a property that does not have a matching request parameter, then the property value is not set.
- **property="propertyName" [param="parameterName"]** : It sets one Bean property to the value of one request parameter. In the syntax, **property** specifies the name of the Bean property and **param** specifies the name of the request parameter by which data is being sent from the client to the server. If the Bean property and the request parameter have different names, then we must specify both property and param. If they have the same name, then we can specify **property** and omit **param**. If a parameter has an empty or null value, then the corresponding Bean property is not set.
- **property="propertyName" value="{string | <%= expression %>}"** : It sets one Bean property to a specific value. The value can be a String or an expression that is evaluated at runtime. We cannot use both the param and value attributes in a <jsp:setProperty> element.



Bean Related Actions:

Type Conversion for <jsp:setProperty> Action:

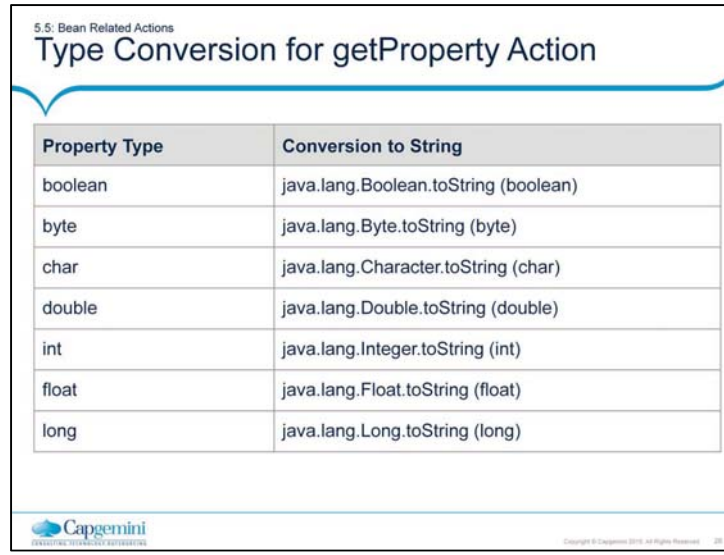
- All property setter methods accessed with a **<jsp:setProperty>** tag will be automatically converted from a String to the appropriate native type by the JSP container. This is accomplished via methods of Java's wrapper classes, as shown in the table in the above slide.
- A conversion failure leads to an error. The error may be at **translation** or at **request-time**.



Bean Related Actions:
The jsp:getProperty Action:
The jsp:getProperty element retrieves the value of a bean property, converts it to a string, and inserts it into the output. The two required attributes are:

name: the name of a bean previously referenced via jsp:useBean
property: the property whose value should be inserted

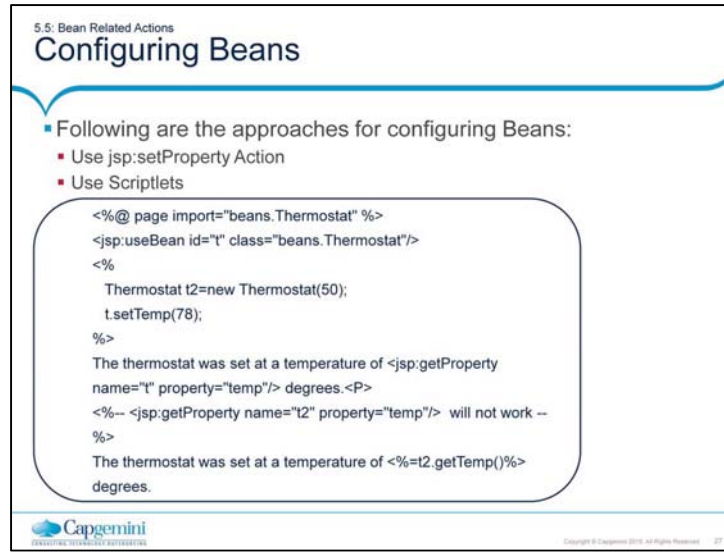
The syntax is very simple and is listed in the above slide. We will need to specify the Bean instance name (should match with the id specified in the jsp:useBean action) and the property name.
Internally this action invokes the getter method of the specified property and converts the return value into string.



Bean Related Actions:

Type Conversion for jsp:getProperty Action

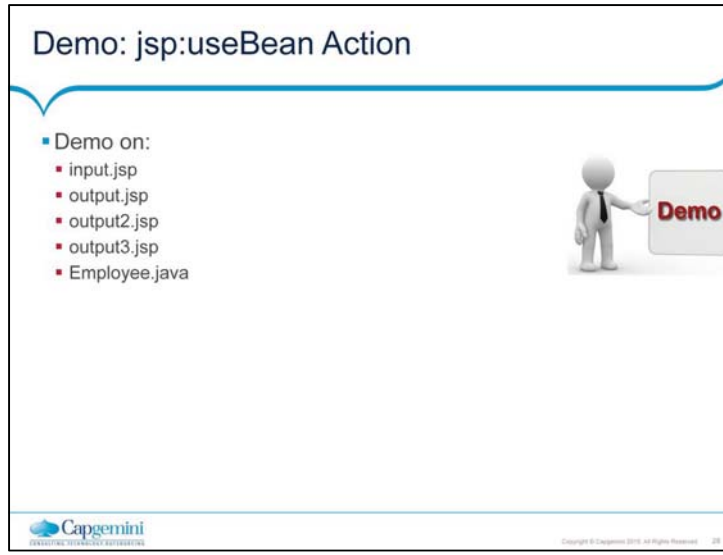
- A JSP component's properties are not limited to string values, but it is important to understand that all property values accessed through the **<jsp:getProperty>** tag will be converted to strings. A **getter** method need not return a String explicitly. However, the JSP container will automatically convert the return value to a String as needed.
- For the Java primitive types, conversion is handled by the methods shown in Table in the above slide.
- Properties are not restricted to primitive types either. For objects, the JSP container will invoke the object's **toString()** method, which unless overloaded it in the class, will probably not be very representative of the data stored in the object.
- We can also overload **getter** and **setter** methods to accept the appropriate object type, although custom tags or JSP scripting elements will be required to access the overloaded methods, since the <jsp:setProperty> and <jsp:getProperty> tags work exclusively with String values.



Bean Related Actions:

Configuring Beans:

- Many times a Bean requires runtime configuration by the page that is initializing it before it can properly perform its tasks. Since we cannot pass information into the Bean's **constructor**, we have to use the **Bean's properties** to hold configuration information. We do this by setting the appropriate **property values** immediately after the container instantiates the Bean in the body of the **<jsp:useBean>** tag or anywhere in the page before the **Bean's properties** are accessed. It can be useful to set a flag in class to indicate whether an instance is in a useful state, toggling the flag when all of the necessary propearties have been set.
- Even though the **Bean tags** do not allow to pass any arguments into a **Bean's constructor**, we can still define constructors that take arguments. We will not, however, be able to call them through **Bean tags**. The only way to instantiate an object requiring arguments in its constructor within a JSP page is through a **scriptlet**.
- One technique that has been found useful is to provide a single method that handles all the configuration steps. This method can be called by the **constructors** that take arguments, for use outside the Bean tags, as well as by the **property access methods** once all the necessary properties have been configured.



Refer Demo on **Lesson5-JSPActions, useBean** folder and show demo by executing each of the above JSP pages.
Note: The listing for **Employee.java (Bean)** is as follows:

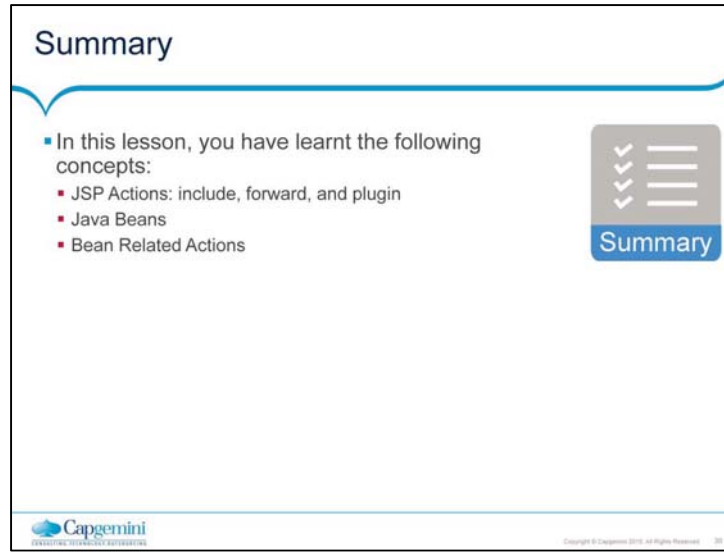
```
public class Employee {
private int eid;
private String enm;
private double esl;

public int getEid() {
return eid;
}
public void setEid(int eid) {
this.eid = eid;
}
public String getEnm() {
return enm;
}
public void setEnm(String enm) {
this.enm = enm;
}
public double getEsl() {
return esl;
}
public void setEsl(double esl) {
this.esl = esl;
}
}
```

Lab

- Lab 2.1
- Lab 2.2

Capgemini


Summary
In this lesson, you have learnt the following concepts:
JSP Actions: include, forward, and plugin
Java Beans
Bean Related Actions
Summary
Capgemini

Review Question

- Question 1: The include action is a ___ include.
  - Option 1: dynamic
  - Option 2: static
  - Option 3: both the above
- Question 2: The ___ attribute of include action is mandatory to be set.
- Question 3: ___ sub-element can be used to pass information to included or forwarded JSP page.
- Question 4: ___ action terminates the execution of current page before moving on to the next page.

Capgemini

Review Question

- Question 5: ___ is a client-side redirect.
- Question 6: ___ action generates browser dependent construct, namely object or embed.
- Question 7: ___ tag is used to display message for the user if the plugin cannot be started.
- Question 8: Java Bean does not extend any specific base class or implement any interface. True/False
- Question 9: JSP uses ___ to interact and access Java Bean's and its properties.

Capgemini

Review Question

- Question 10: The useBean action creates a new instance of a bean only if an existing instance of the same bean within the scope is not available. True/False
- Question 11: The conversion from string value to appropriate data type is done by ___ action.
- Question 12: The multiple argument constructor of a Java Bean can be called through JSP actions in a JSP page.
  - True/False

Capgemini

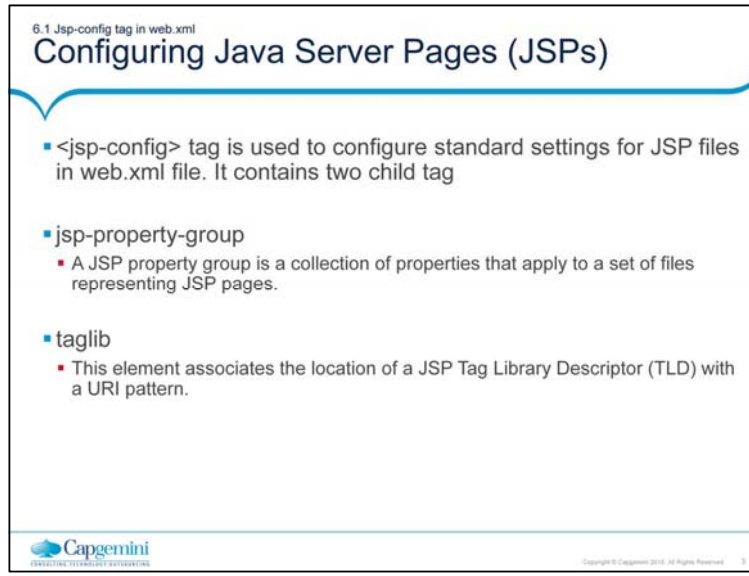
Java Server Pages

Lesson 6: JSP Configuration in Web.xml

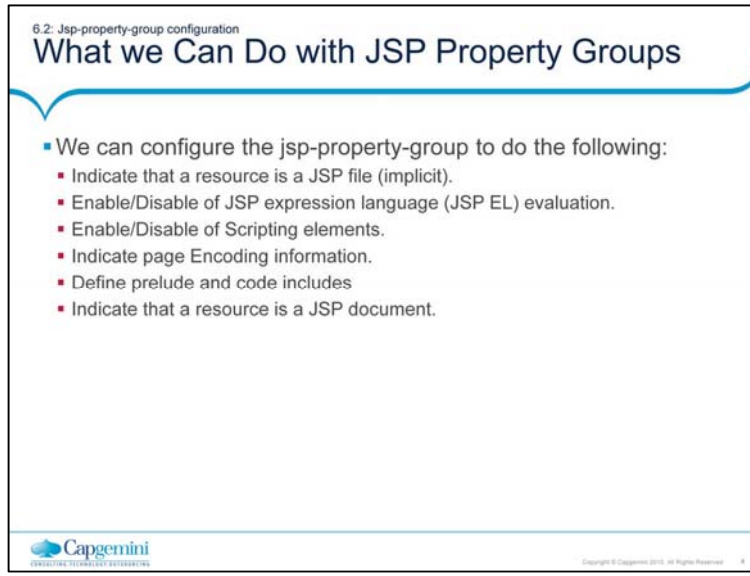
Lesson Objectives

- In this lesson, you will learn:
  - Jsp-config tag in web.xml
  - Jsp-property-group configuration

Capgemini

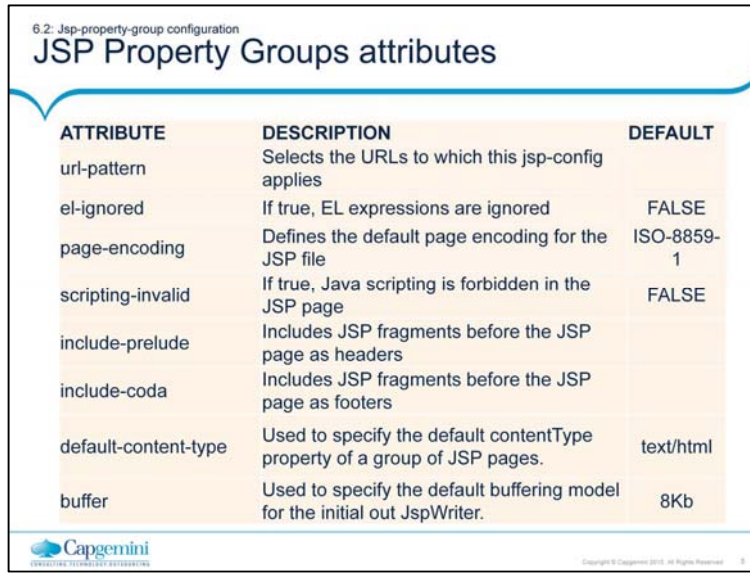


<jsp-config> added in JSP2.0 in JEE4 specification.
taglib will be discussed in subsequent lesson of Custom Tags.

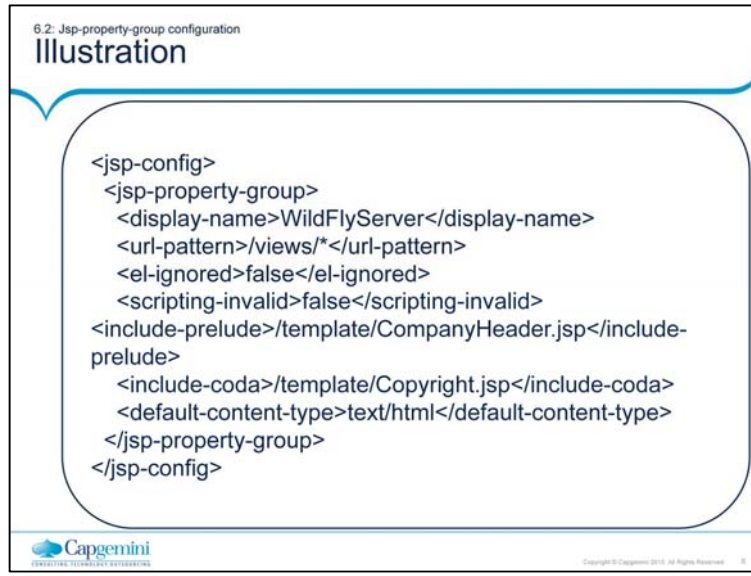


We can configure a number of properties for a group of JSP pages through jsp-property-group tag in web.xml file. Such as enable or disable EL evaluation, enable or disable scripting elements, specify page encoding etc.

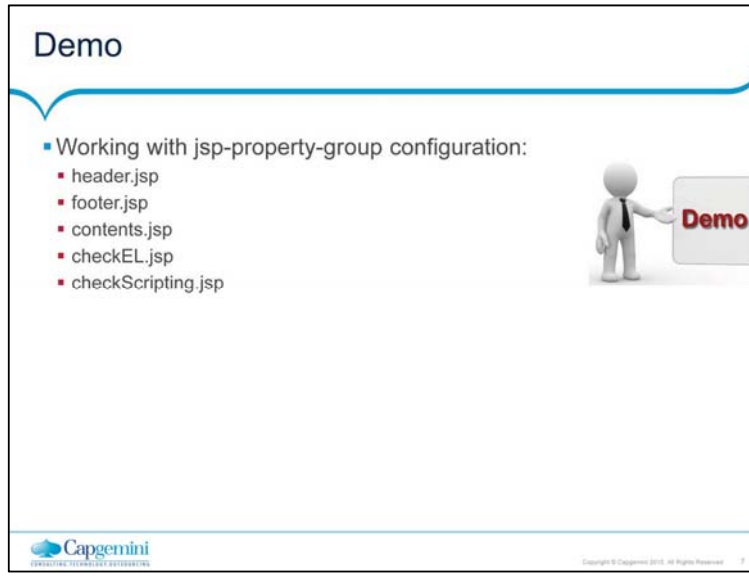
We can implicitly include preludes (also called headers) and codas (also called footers) for a group of JSP pages. When the elements are present, the given paths are automatically included (as in an include directive) at the beginning and end of each JSP page in the property group respectively.



url-pattern : selects the URLs which this jsp-config applies to a group of jsp pages.
el-ignored : If true, EL expressions are ignored in the group of jsp pages specified in url-pattern.
page-encoding : Defines the default page encoding for the JSP file.
scripting-invalid : If true, Java scripting is forbidded in the group of jsp pages specified in url-pattern.
include-prelude : Includes JSP fragments before the group of jsp pages specified in url-pattern as headers.
include-coda : Includes JSP fragments before the group of jsp pages specified in url-pattern as footers
default-content-type : (Added in JSP2.2) Used to specify the default contentType property of a group of JSP pages.
Buffer : (Added in JSP2.2) Used to specify the default buffering model for the initial out JspWriter for the group of jsp pages specified in url-pattern.



In the above illustration, properties are applied to the JSP pages in the views folder. So, all JSP pages in that folder will include CompanyHeader.jsp as header and Copyright.jsp as footer.

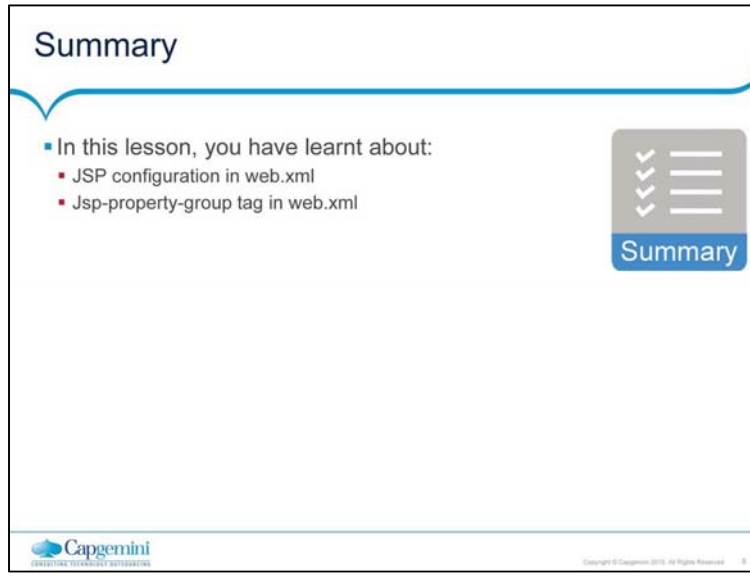


Deploy web application **Lesson6-JSP_Config** and show demo by executing each of the above JSP pages.

Lab: JSP property group configuration

- Lab 2.3

Capgemini


Summary
In this lesson, you have learnt about:
JSP configuration in web.xml
Jsp-property-group tag in web.xml
Summary
Capgemini

Review Question

- Question 1: Which of the following tag can not be defined inside jsp-property-group tag?
  - Option 1: el-ignored
  - Option 2: buffer
  - Option 3: jsp-config
  - Option 4: is-xml

- Question 2: Which of the following tag is used to ignored EL expression in JSP page group ?
  - is-el-ignored
  - el-ignored
  - ignore-el

Capgemini

Review Question

- Question 3: Which of the following tag provide the JSP group url ?
  - Option 1: jsp-url
  - Option 2: url-param
  - Option 3: url-pattern
  - Option 4: url-group

Capgemini

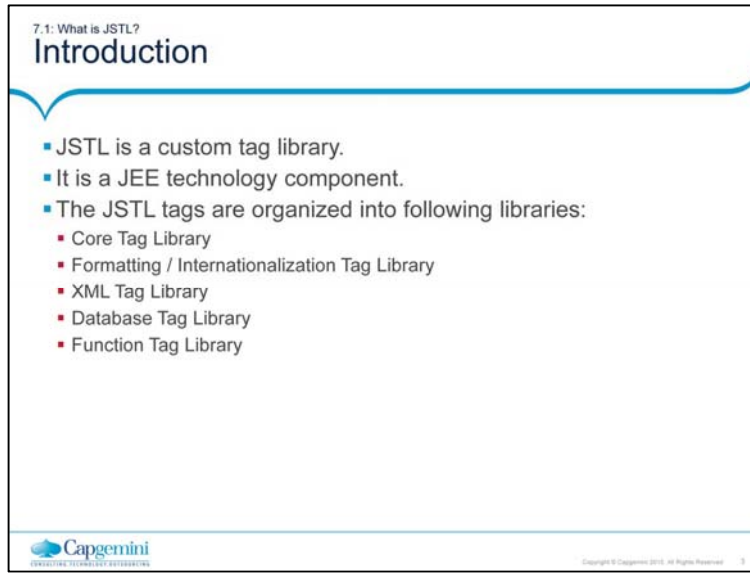
Java Server Pages

Lesson 7: JSP Standard Tag Library (JSTL)

Lesson Objectives

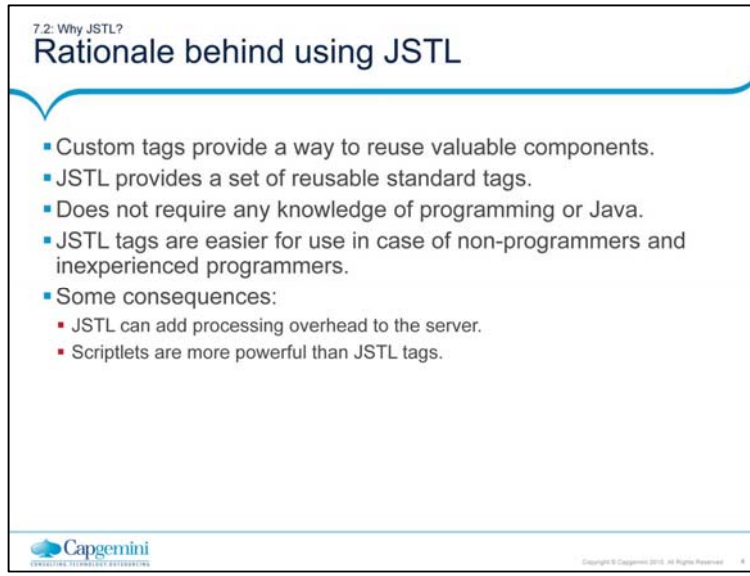
- In this lesson, you will learn:
  - What is JSTL?
  - Why JSTL?
  - Using Expression Language
    - EL Capabilities
    - Accessing Scoped Variables
    - Accessing Bean Properties
    - Accessing Collections
    - Accessing Implicit Objects
    - Using Lambda in EL
    - EL Operators
  - Using JSTL
    - Working with Core Tags

Capgemini



What is JSTL?

- **JSTL (JSP Standard Tag Libraries)** is a collection of JSP custom tags.
- The goal of JSTL, as described in the specification, is to help simplify JavaServer Pages page authors' lives.
- To achieve this goal, JSTL has provided custom tags for many common JSP page authoring tasks that require scripting statements to manipulate server side dynamic data.
- JSTL has tags such as iterators and conditionals for handling flow control, tags for manipulating XML documents, internationalization tags, tags for accessing databases using SQL, and commonly used functions.
- JSTL offers tags through following libraries:
  - **core:** Basic scripting functions
  - **xml:** XML processing
  - **fmt:** Internationalization of formatting
  - **sql:** Data base accessing
  - **fn:** JSTL functions

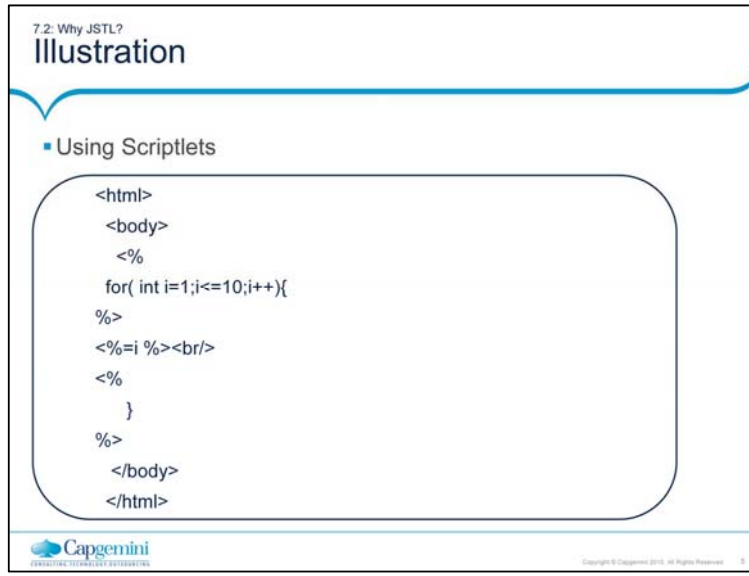


Why JSTL?

- Scriptlets are hard to read and hard to maintain. JSP was improved throughout its history, so that even people who do not know Java well (or have a limited knowledge of it) can make presentation pages.
- JSTL can help avoiding the use of scripting elements in JSP pages. As a matter of practice, a JSP page should be a view page, it should not have Java code embedded in **scriptlets**. This aids in separation of concerns.
- Custom tags provide a way to reuse valuable components.
- JSTL provides a set of *reusable* standard tags.
- JSTL tags can be used easily and effectively by non-programmers and inexperienced programmers. This is because they do not require any knowledge of programming or Java.
- Since JSTL tags are XML, they cleanly and uniformly blend into a page's HTML markup tags.

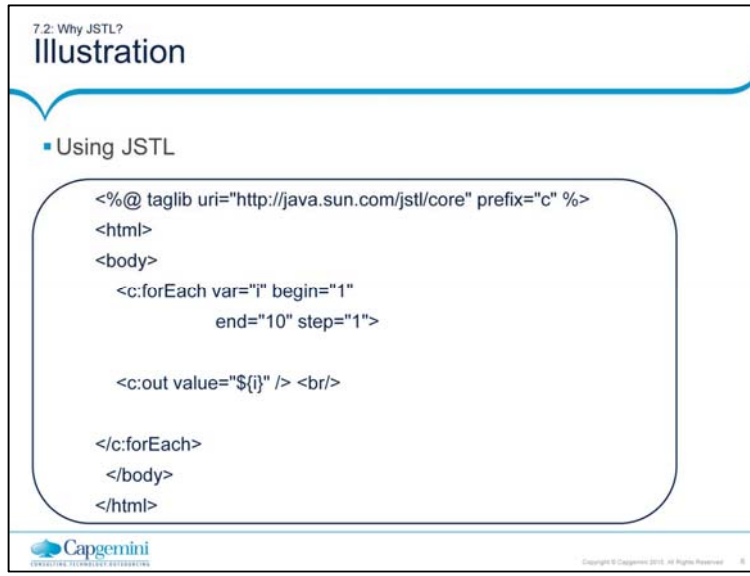
JSTL Consequences:

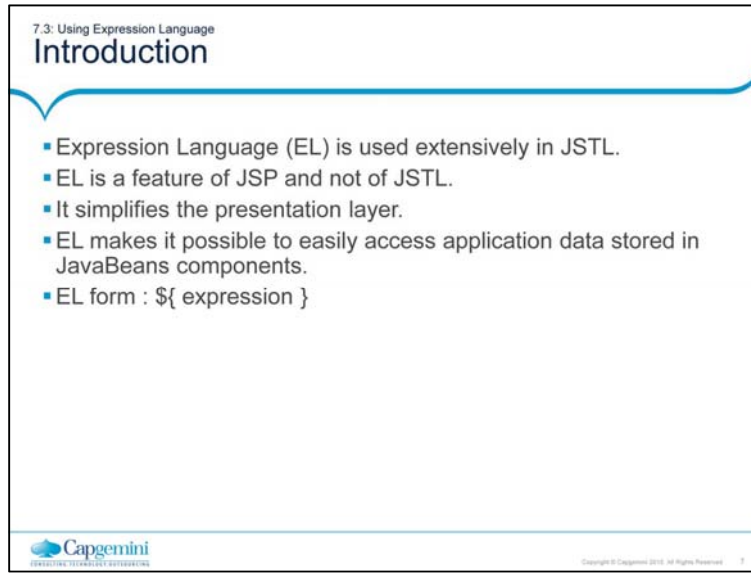
1. JSTL can add processing overhead to the server.
    - Java code embedded in scriptlet tags is pretty much just copied into the resulting servlet. JSTL tags, on the other hand, cause much more code to be added to the servlet.
2. Scriptlets are more powerful than JSTL tags.
    - Although JSTL provides a powerful set of reusable libraries, it cannot do everything that Java code can do. It is designed to facilitate scripting the presentation code.



Why JSTL?

- The above slide illustrates the JSP page with scriptlets and JSP page with JSTL. The JSP page with scriptlets contains the Java code embedded within JSP scripting tags. This requires an expert in Java programming.
- On the other hand, the JSP page with JSTL contains only tags and no Java code. So page authors (Non-Java programmers) can also develop the JSP pages using JSTL. The above example uses “ForEach” tag from core JSTL tag library which replaces the Java for loop.


7.2: Why JSTL?
Illustration
Using JSTL
<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>
<html>
<body>
<c:forEach var="i" begin="1"
end="10" step="1">
<c:out value="${i}" />
</c:forEach>
</body>
</html>
Capgemini

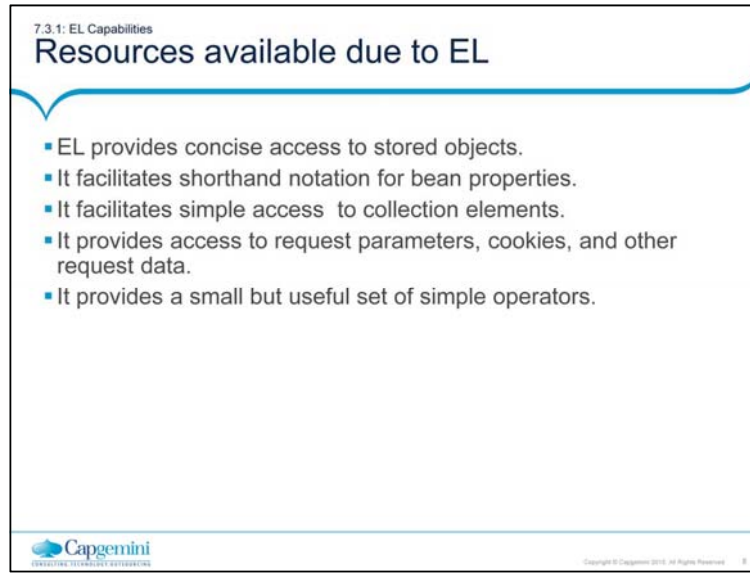


Using Expression Language:

- Expression Language is introduced with JSP 2.0, and many more capabilities are added with JSP 2.1.
- Expression Language is used with JSTL to simplify the presentation layer.
- Expression Language replaces the action tags <jsp:useBean> <jsp:getProperty> used to access Java beans properties with short and readable expressions.
- Express Language uses the form **${expr}** to access and specify an expression.

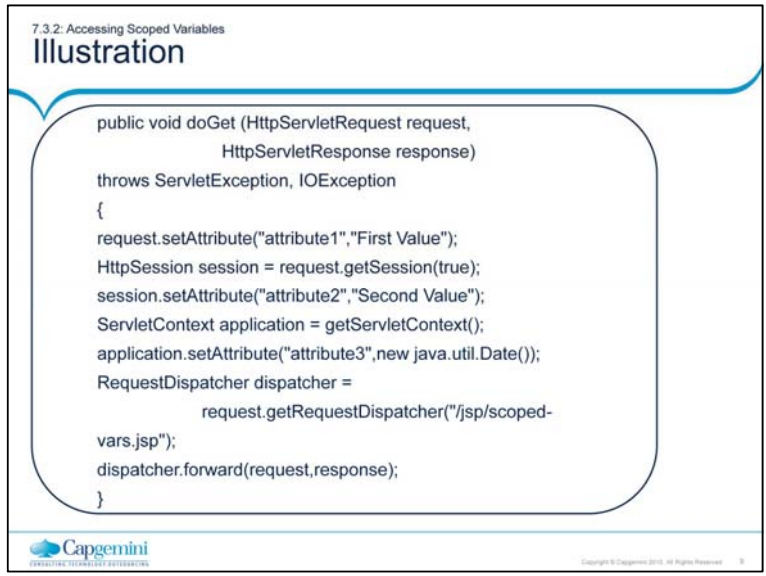
Example:

- ${rs.rowCount}, ${x+y}



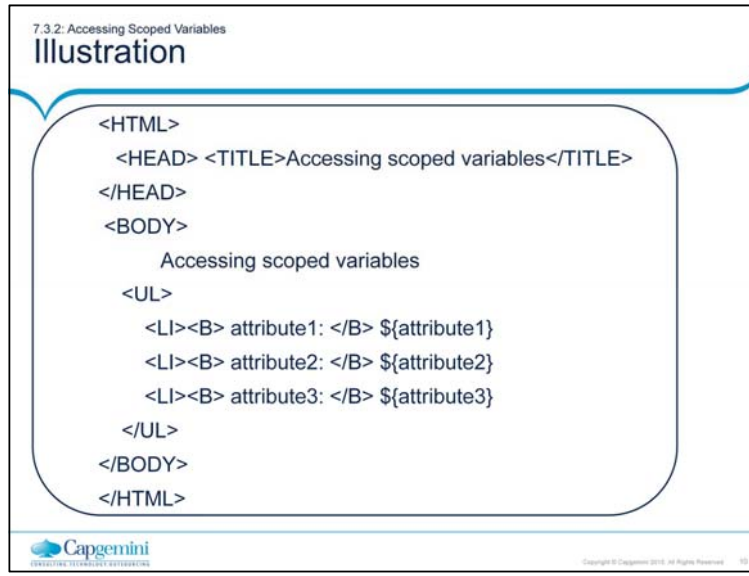
Expression Language Capabilities:

- EL provides concise access to stored objects: to output a “scoped variable” named saleItem, use ${saleItem}
- It facilitates shorthand notation for bean properties: To output the companyName property of a scoped variable named company, use ${company.companyName}.
- It facilitates simple access to collection elements: To access an element of an array, List or Map, use ${variable[indexorKey]}
- It provides access to request parameters, cookies, and other request data: To access the standard types of request data, we can use one of the implicit objects mentioned in first lesson.
- It provides a small but useful set of simple operators.



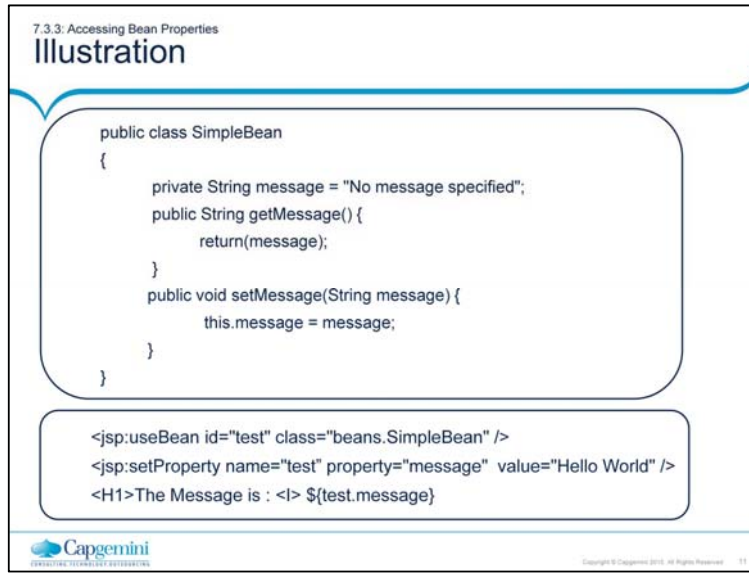
Accessing Scoped Variables:

- The code in the above slide defines different variables in different scopes and sets the values to these variables. There are various scopes like request, session, and application in which variables are set.
- Furthermore, this servlet forwards request to **scoped-vars.jsp** page. This jsp page uses expression language to retrieve these variables. The code snippet for that is given in the next slide.

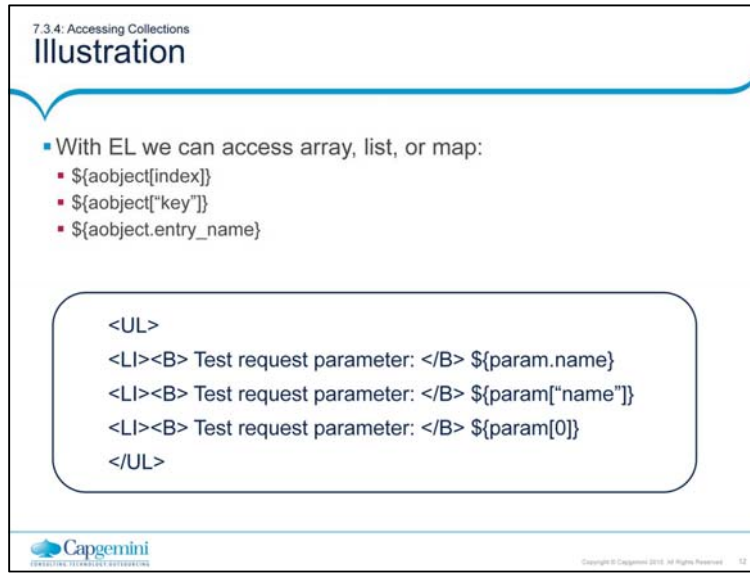


Accessing Scoped Variables:

- The above slide shows the code from **scoped-vars.jsp** page content. By using expression language the scoped variables defined in the previous servlet are accessed here.
- **Example:**
  ${attribute1} retrieves the value of scoped variable “attribute1”.



Accessing Bean Properties:
The illustration in the above slide is accessing the "**SimpleBean**" bean property "**message**" from JSP page using EL statement ${test.message}.



Accessing Collections:

- Using EL, we can access the collection elements, as well. The illustration is given in the above slide.
- The EL allows accessing collections using array notation.
  For example: If **AttributeName** is scoped variable and refers to an array, list or map, then we can access the collection with the following:

```
${AttributeName[entryname]}
```

- If the scoped variable is an array and entryname is the index, then an value is obtained with **array[index]**.
  For example: If CustomerNames refers to an array of strings, then the following entry will output first entry:

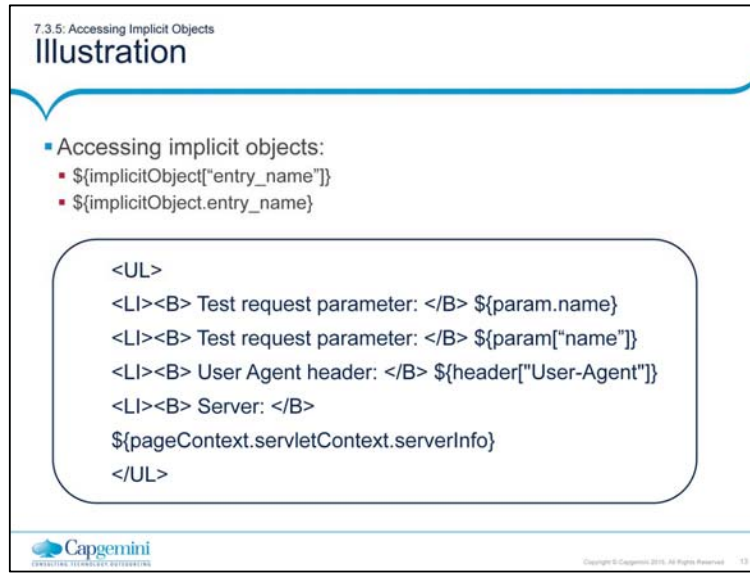
```
${CustomerNames[0]}
```

  Similarly, the entries from a List can be output, as well.

- For Map, entry name is key and value is obtained with Map.get(key). In EL, the following example shows how to retrieve value from a map.

```
${stateCapitals["Maryland"]}
```

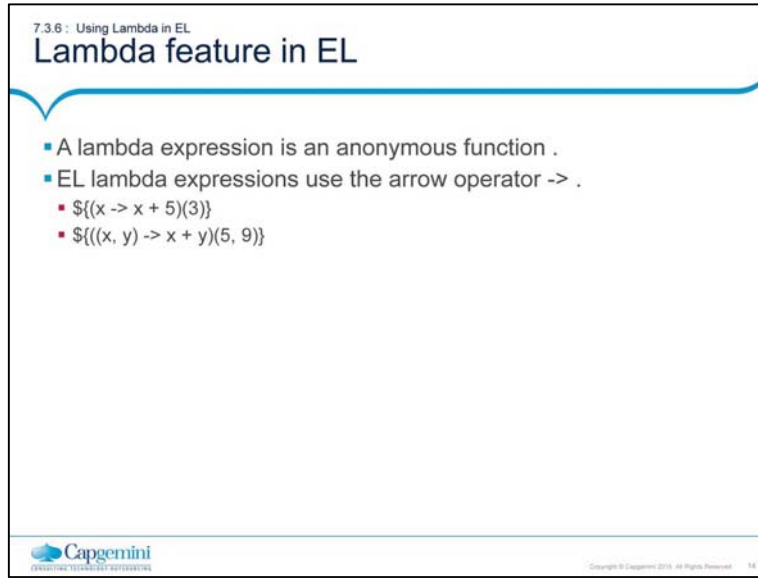
- The above example will return Annapolis.



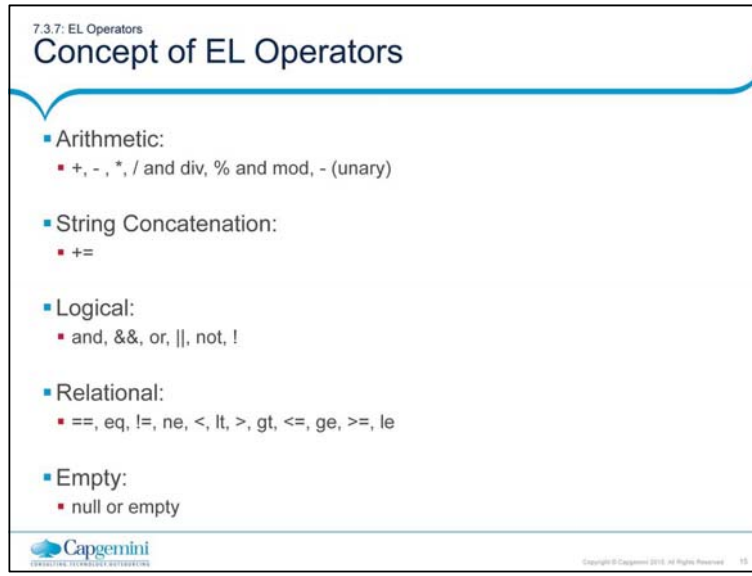
Accessing Implicit Object:

The implicit objects provided by an JSP page can be used as follows:

- **PageContext:** For example, ${pageContext.session.id} returns the current session ID.
- **Param and paramValues:** These objects allows to use the primary request parameter value (param) or array of parameter values (paramValues). For example, ${param.custID} returns the value of custID request parameter, or null if it does not exist.
- **header and headerValues:** These access the HTTP request header values. For example: ${header.accept} accesses the accept header
- **initParam:** It allows to access context initialization parameters. For example: ${initParam.defaultColor} accesses init parameter called defaultColor.
- **pageScope, requestScope, sessionscope, applicationScope :** These objects restrict the locations where system looks up for scoped variables. For Example, ${name} will search for name in all scopes, whereas ${requestScope.name} will search for name only in HttpServletRequest.

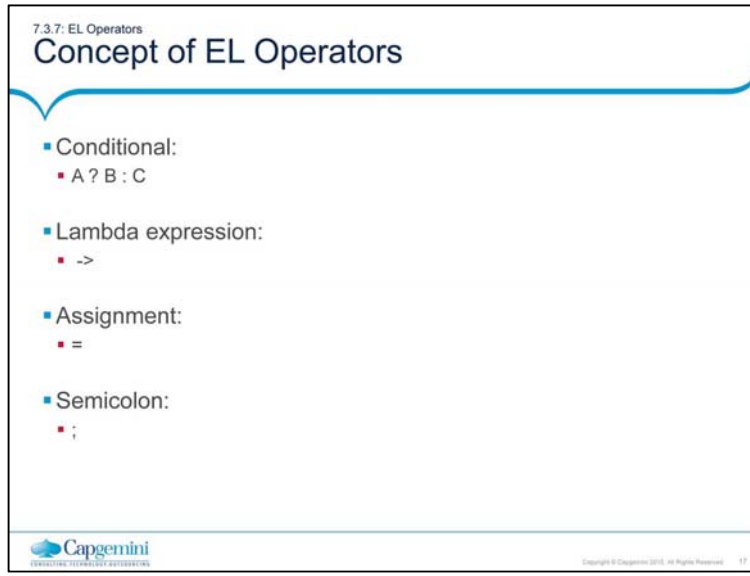


A lambda expression is an anonymous function that, typically, is passed as an argument to a higher-order function (such as a Java method). In the most general sense, lambda expressions are a list of parameter names (or some placeholder if the function has no parameters), followed by some type of operator, and finally the function body.
The primary difference between the Java 8 and EL lambda expressions is that in Java the body of a lambda expression can contain anything that's legal in a Java method, whereas in EL the body of a lambda expression is another EL expression. EL lambda expressions use the arrow operator -> to separate the expression parameters on the left side from the expression in the right side. Also, again as with Java lambda expressions, the parentheses around the expression parameter are optional if there is exactly one parameter.${(x -> x + 5)(3)}${((x, y) -> x + y)(5, 9)}


7.3.7: EL Operators
Concept of EL Operators
Arithmetic:
+, - , *, / and div, % and mod, - (unary)
String Concatenation:
+=
Logical:
and, &&, or, ||, not, !
Relational:
==, eq, !=, ne, <, lt, >, gt, <=, ge, >=, le
Empty:
null or empty
Capgemini

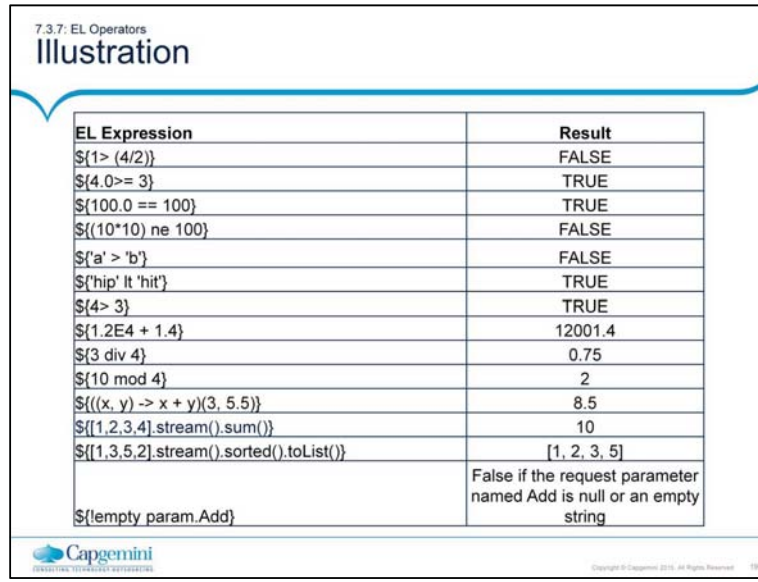
EL Operators:
The JSP expression language provides the following operators:
Arithmetic: +, - (binary), *, / and div, % and mod, - (unary).
String concatenation: +=.
Logical: and, &&, or, ||, not, !.
Relational: ==, eq, !=, ne, <, lt, >, gt, <=, ge, >=, le. Comparisons can be made against other values or against Boolean, string, integer, or floating-point literals.
Empty: The empty operator is a prefix operation that can be used to determine whether a value is null or empty.
Conditional: A ? B : C. Evaluate B or C, depending on the result of the evaluation of A.
Lambda expression: ->, the arrow token.
Assignment: =.
Semicolon: ;.

The precedence of operators highest to lowest, left to right is as follows:
The precedence of operators, highest to lowest, left to right, is as follows:
[] .
() (used to change the precedence of operators)
- (unary) not ! empty
* / div % mod
+ - (binary)
+=
<> <= >= lt gt le ge
== != eq ne
&& and
|| or
? :
->
=
;


7.3.7: EL Operators
Concept of EL Operators
Conditional:
A ? B : C
Lambda expression:
->
Assignment:
=
Semicolon:
;
Capgemini

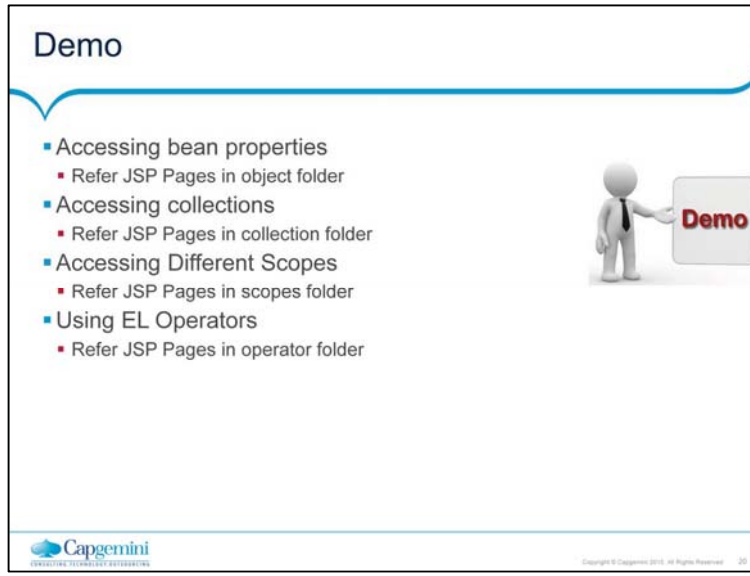
EL Operators:
The JSP expression language provides the following operators:
Arithmetic: +, - (binary), *, / and div, % and mod, - (unary).
String concatenation: +=.
Logical: and, &&, or, ||, not, !.
Relational: ==, eq, !=, ne, <, lt, >, gt, <=, ge, >=, le. Comparisons can be made against other values or against Boolean, string, integer, or floating-point literals.
Empty: The empty operator is a prefix operation that can be used to determine whether a value is null or empty.
Conditional: A ? B : C. Evaluate B or C, depending on the result of the evaluation of A.
Lambda expression: ->, the arrow token.
Assignment: =.
Semicolon: ;.

The precedence of operators highest to lowest, left to right is as follows:
The precedence of operators, highest to lowest, left to right, is as follows:
[] .
() (used to change the precedence of operators)
- (unary) not ! empty
* / div % mod
+ - (binary)
+=
<> <= >= lt gt le ge
== != eq ne
&& and
|| or
? :
->
=
;

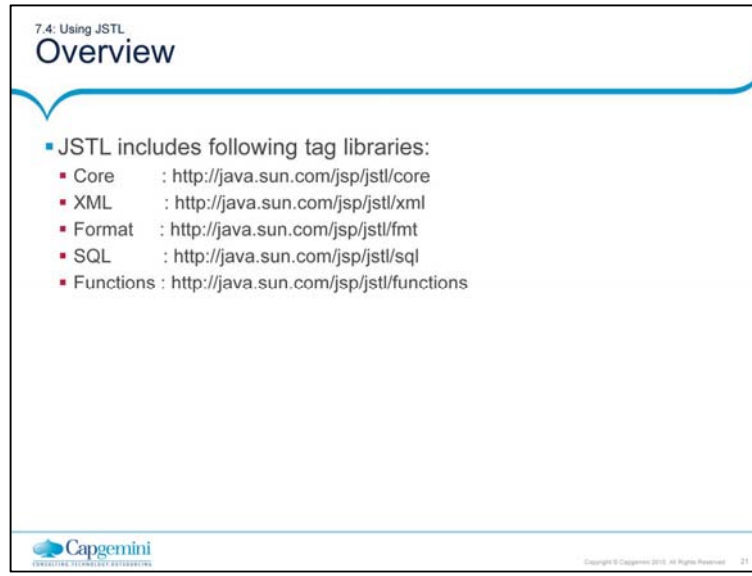


Note : For the following example in JSP page java.util.stream must be imported which is included in Java8:
[1,2,3,4].stream().sum()
[1,3,5,2].stream().sorted().toList()

Java.util.stream package contains the classes to support functional-style operations on streams of elements, such as map-reduce transformations on collections.

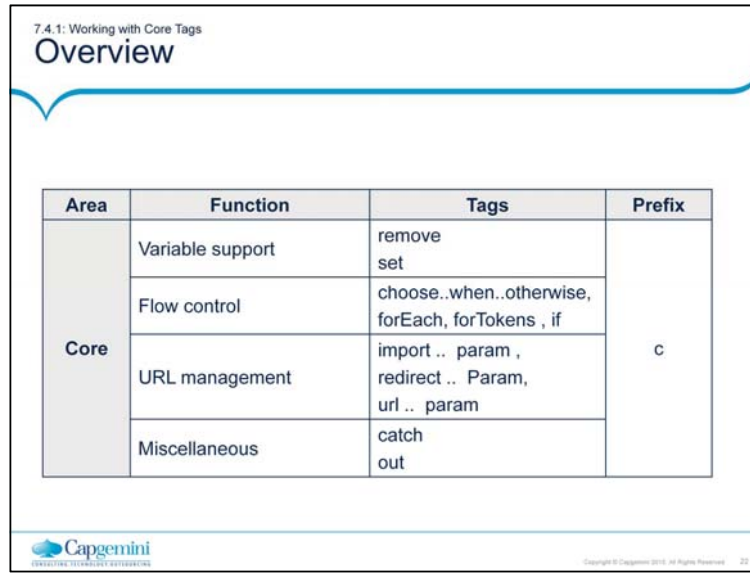


Deploy web application **Lesson7-DemoEL** and show demo by executing each of the above JSP pages in corresponding folder.



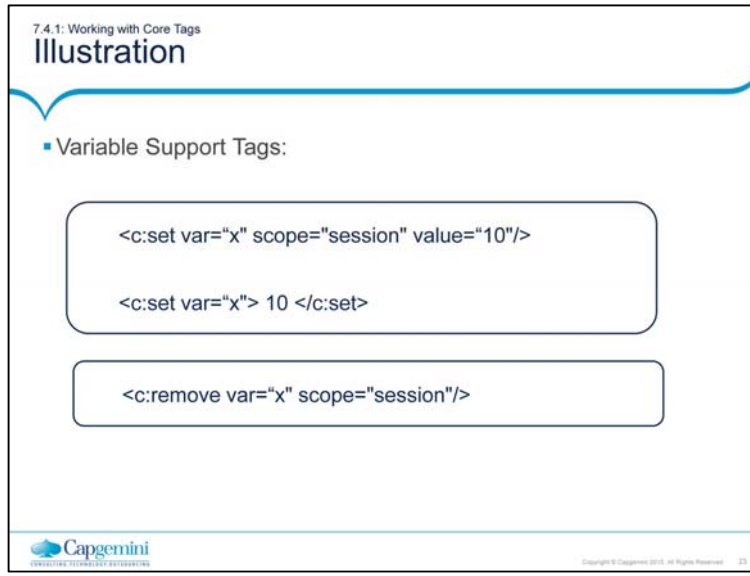
Using JSTL:

- The **JavaServer Pages Standard Tag Library (JSTL)** encapsulates core functionality common to many JSP applications. Instead of mixing tags from numerous vendors in JSP applications, JSTL allows to employ a single, standard set of tags. This standardization allows to deploy applications on any JSP container supporting JSTL and makes it more likely that the implementation of the tags is optimized.
- JSTL has tags such as iterators and conditionals for handling flow control, tags for manipulating XML documents, internationalization tags, tags for accessing databases using SQL, and commonly used functions.



Working with Core Tags:

- Core tags provide tags related to variables and flow control as well as a generic way to access URL-based resources whose content can then be included or processed within the JSP page.
  - **<c:set>:** It specifies basic variable setting actions
  - **<c:out>:** It specifies basic output actions
  - **<c:if test= ? :** Conditional action
  - **<c:choose>, <c:when> , <c:otherwise>:** Conditional action
  - **<c:forEach> :** Iteration actions
  - **<c:forTokens> :** Iteration actions
  - **<c:import> :** It retrieves content from a local jsp page or another server.
  - **<c:url> :** It specifies the URL of the content to retrieve.
  - **<c:redirect> :** It sends a HTTP redirect to the client.
  - **<c:param> :** It adds request parameters to an URL.
  - **<c:catch> :** It catches a java.lang.Throwable thrown by any of its nested actions.
- Next few slides illustrate these tags.



Working with Core Tags:

Variable Support Tags:

- The set tag sets the value of an EL variable or the property of an EL variable in any of the JSP scopes (page, request, session, or application). If the variable does not already exist, it is created.
- The JSP EL variable or property can be set from the attribute value as shown below:

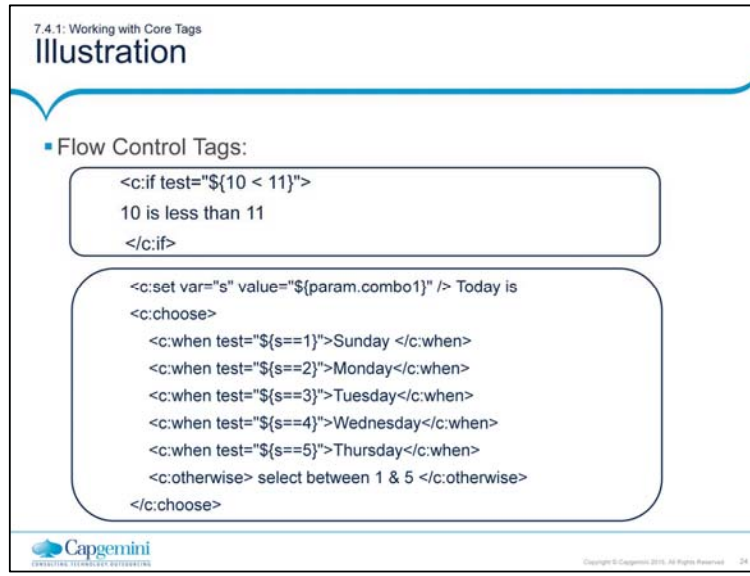
```
<c:set var="x" scope="session" value="10"/>
```

- Alternatively it can be set from the body of the tag, as shown below:

```
<c:set var="x">
  10
</c:set>
```

- To remove an EL variable, use the remove tag. Above variable is removed as follows:

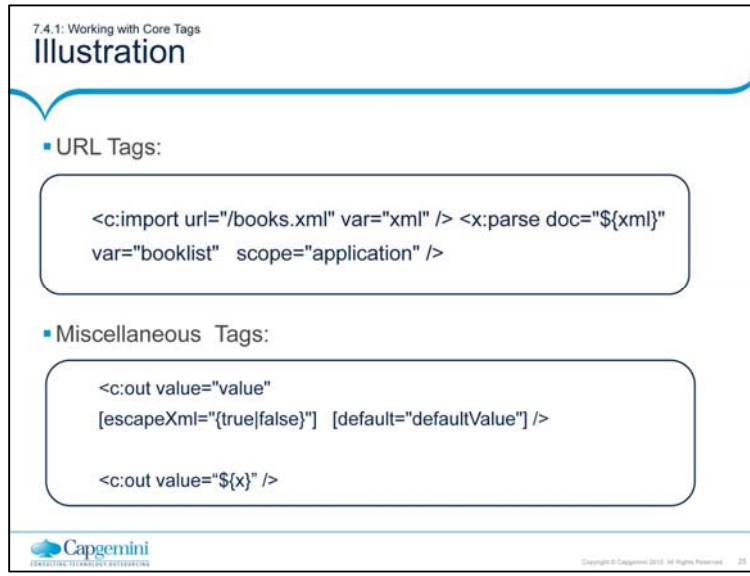
```
<c:remove var="x" scope="session"/>
```



Working with Core Tags:

Flow Control Tags :

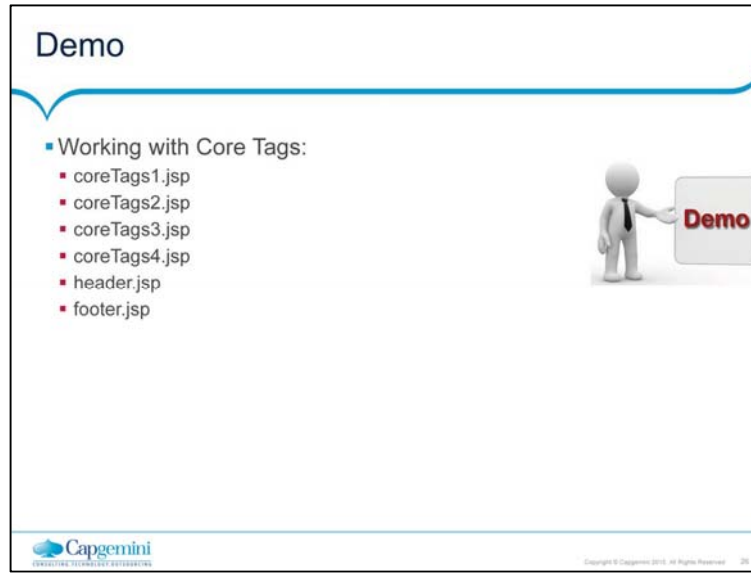
- The illustration in the above slide shows the use of conditional tags like <c:if> and <c:choose>.
- For iteration tag illustration, please refer the slide no-6.



Working with Core Tags:

URL and Miscellaneous Tags:

- In the illustration in the above slide, "import" is used to read the XML document containing book information and assign the content to the scoped variable xml.
- The out tag evaluates an expression and outputs the result of the evaluation to the current **JspWriter** object.

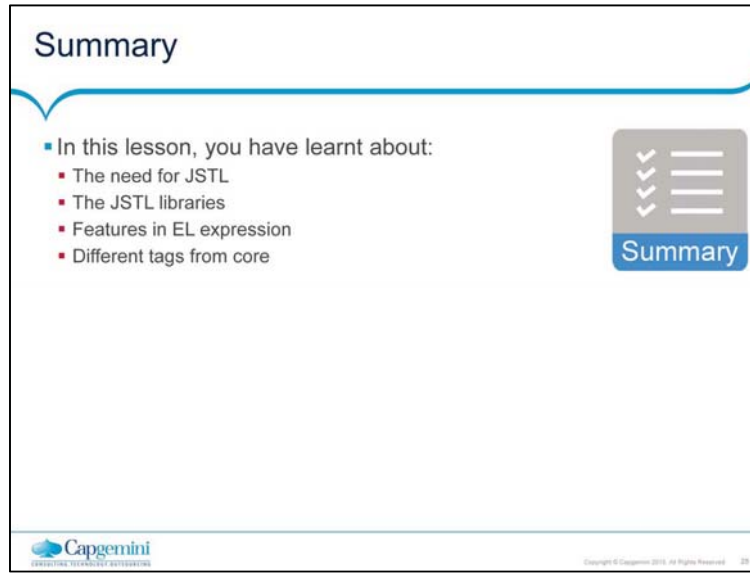


Deploy web application **Lesson7-JSTL_Core_Tags** and show demo by executing index.jsp and referring to each of the above JSP pages.

Lab: JSTL

- Lab 3.1
- Lab 3.2
- Lab 3.3

Capgemini


Summary
In this lesson, you have learnt about:
The need for JSTL
The JSTL libraries
Features in EL expression
Different tags from core
Summary
Capgemini

Review Question

- Question 1: Which of the following library provides tags for internationalization?
  - Option 1: sql
  - Option 2: xml
  - Option 3: fmt
  - Option 4: core

- Question 2: EL expression can be used within JSP scriptlet tags.
  - True/False

Capgemini

Review Question

- Question 3: Which of the following library provides tags for adding request parameters to an URL
  - Option 1: sql
  - Option 2: fn
  - Option 3: fmt
  - Option 4: core

- Question 4: EL language uses '#' symbol to write expression.
  - True / False

Capgemini

Java Server Pages v2.2

Lab Book

Document Revision History

Date	Revision No.	Author	Summary of Changes
Nov-2009	2.0	PradnyaJagtap, Habib Shaikh, and Mahima Sharma	Revamped from JSP 2.0 to JSP 2.1
25-Nov-2009		CLS Team	Review Template application
Feb-2010	2.1	Mahima Sharma	Refined based on feedback received from faculties.
Sep-2012	2.2	Sarbananda Behera	Refined to make problem based lab exercises
Sep - 2014	2.3	Anjulata	Refinement in problem statements.
May-2015	2.4	Anjulata	Revamped from JSP 2.1 to JSP 2.2
May-2016	2.5	Anjulata	Refinement in problem statements as per new ToC

Table of Contents

Getting Started

Overview

This lab book is an unguided tour for learning Java Server Pages version 2.2. It comprises of problem statements and diagrams to complete a sequences of exercises. Flow diagrams and screen snap shots are provided wherever necessary. It will expose you to create Java Server Pages Applications using eclipse IDE, deploy them on WildFly server and invoke from browser.

Setup Checklist for JSP

Here is what is expected on your machine in order for the lab to work.

Minimum System Requirements

- Intel Core i5 or higher.
- Microsoft Windows 7/8 or higher.
- Memory: 2GB of RAM (more recommended)
- 500MB hard disk space
- VGA or higher resolution monitor
- Mouse or other pointing device
- JDK version 1.8 with help
- Netscape or IE web browser
- Eclipse 4.4(Luna)
- Wildfly Server version 8.x

Please ensure that the following is done:

- JDK 1.8 is installed.
- Wildfly Server is installed.(Wildfly8.x)
- Eclipse 4.4(Luna) or higher version with JEE support with Wildfly8.x Adapter is installed.

Lab 1. **JSP Basics**

Goals	• Create simple JSP programs.
Time	2 hrs.

1.1: Developing Simple JSP Page

Problem:

Develop a simple JSP application to show the current Date and Time from Server.
[Note: Use Java8 Date and Time API]

1.2: JSP Scripting Elements and Method Declarations

Problem:

Create a JSP page to accept username and password from user as shown in figure 1. Authenticate the user using JSP scriptlets and display appropriate success or error message on a new page.

Authentication should be done using a java method that is declaredin the JSP page.

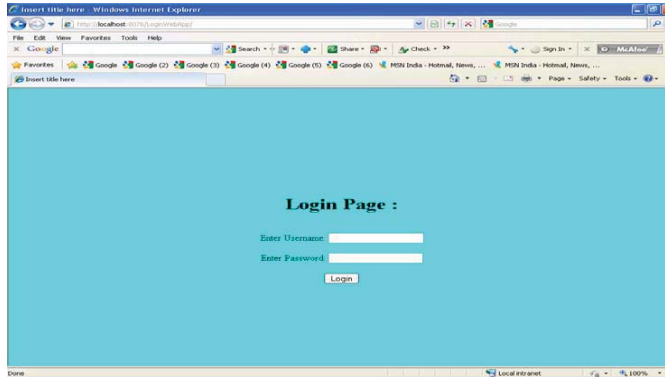


Figure 1: Login.jsp

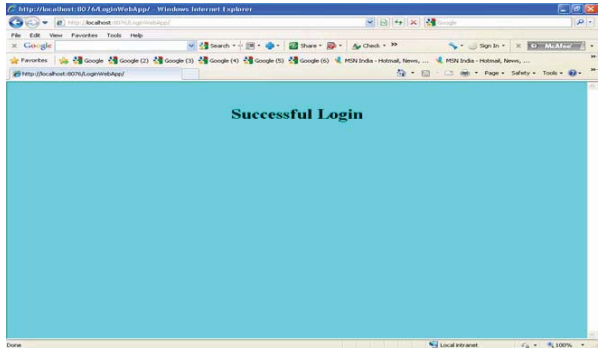


Figure 2: Success.jsp

1.3 : JSP Directive

Problem:

Modify the assignment 5.1 from Servlet Lab book. Covert User_Info page from html to jsp and convert InfoView and ErrorView from servlet to jsp. Give the name of views as Bill_Info.jsp and Error.jsp. Include a common Header page to all these pages with the help of include directive.

In the **Bill_Info.jsp** page set **errorPage** attribute as **"Error.jsp"** and in Error.jsp page set isErrorPage attribute as "true".

In the Bill_Info.jsp page write a scriplet code to check the consumer number.

If consumer number is null then throw an **EBillException** from scriptlet code with a user defined exception message..

Lab 2. **JSP Actions**

Goals	• Using include action to perform a dynamic include • Integrating JSP and Servlets using forward action • Developing Java Beans and interacting with it from a JSP page using bean related actions
Time	3 hours
Pre-requisites	JSP Action tags and Java Beans Concept

2.1: Dynamic Include action

Develop a **header.jsp** page which contains the header section as shown in the figure 3. The contents of the header section are not static. Develop a **home.jsp** page which includes the **header.jsp** page dynamically. The contents of the header section in **header.jsp** should be provided by **home.jsp** page.

Figure 3: Header.jsp page

Note:

The contents of the **header** should not be static. Instead retrieve them from the **request** parameters. The **Home.jsp** page can include the **Header.jsp** page dynamically by using **jsp:include** action and can pass content required by the **header section** of **Header.jsp** page, as well.

Develop the **Home.jsp** page that includes the **Header.jsp** page shown above and also passes the content for **menu_item1** to **menu_item6** as request parameters to build the header dynamically

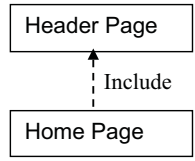


Figure 4

2.2: Integrating Servlet, JSP, and Java Bean (Implementing JSP Model 2 – Model-View-Controller pattern)

Develop an online **E Bill** application that allows the user to accept readings for a consumer and calculate bill amount and persist the details. The work flow of the application is shown in the figure 5 given below:

Note :

1. This is stretched assignment of Servlet assignment 5.1.
2. Use the pages designed in Lab 1.3

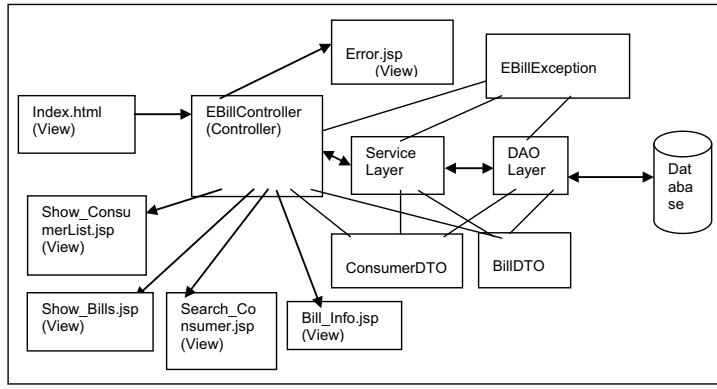


Figure 5: Implementing MVC pattern

Solution:

Follow the **MVC (Model View Controller)** approach wherein a controller servlet will intercept all the requests and select the appropriate model to process the request. After the processing is complete, the controller can forward the request to the appropriate JSP (view) for presentation.

Spec1: Create the following database tables in Oracle

Table Script

```
CREATE TABLE Consumers(
        consumer_num NUMBER(6) PRIMARY KEY,
        consumer_name VARCHAR2(20) NOT NULL,
        address VARCHAR2(30)
);

INSERT INTO Consumers VALUES(100001,'Sumeet','Shivaji Nagar, Pune');
INSERT INTO Consumers VALUES(100002,'Meenal','M G Colony Panvel, Mumbai');

INSERT INTO Consumers VALUES(100003,'Neeraj','Whitefield, Bangalore');
INSERT INTO Consumers VALUES(100004,'Arul','Karapakkam, Chennai');
```



```
CREATE TABLE BillDetails(
        bill_num NUMBER(6) PRIMARY KEY,
        consumer_num NUMBER(6) REFERENCES Consumers(consumer_num),
        cur_reading NUMBER(5,2),
        unitConsumed NUMBER(5,2),
        netAmount NUMBER(5,2),
        bill_date DATE DEFAULT SYSDATE);

CREATE SEQUENCE seq_bill_num START WITH 100;
```

Spec2: Create the view **index.html**as shown in following figure 6.

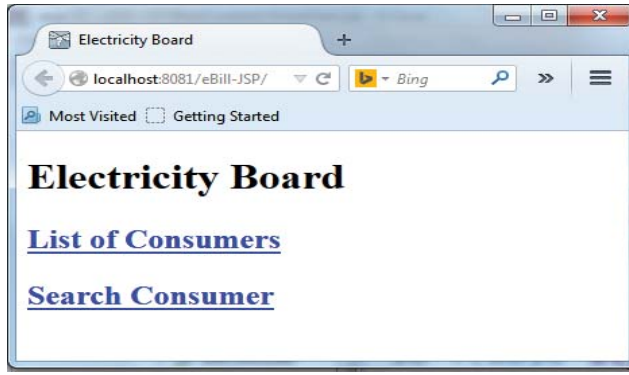


Figure 6:Index.html

Spec3: Develop the controller **EBillController**that receive the request received from the **Index.html**anddo the appropriate processing.

It will display page as shown in thefollowing figure 7forfirst link:

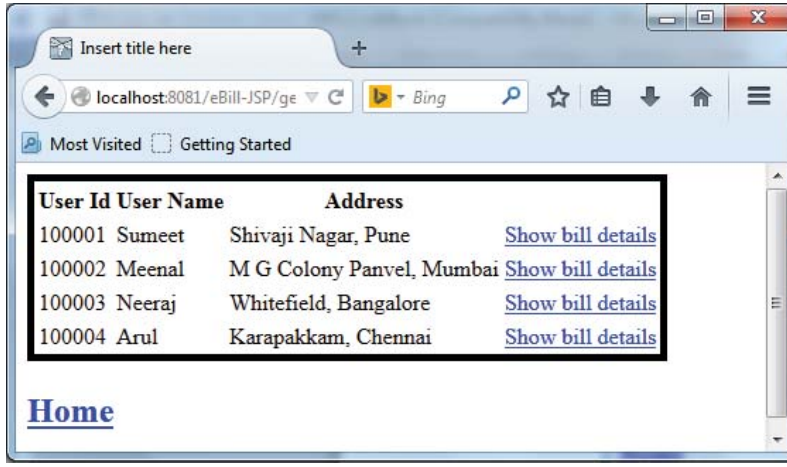


Figure 7: Show_ConsumerList.jsp

And display page as shown in the figure 8 for second link from index page:

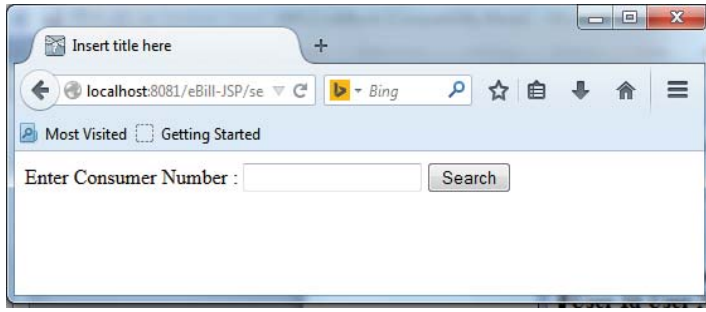


Figure 8: Search_Consumer.jsp

Spec 4: After clicking on the Search button for a valid user id in **Search_Consumer** pageshown in the figure 9 should be displayed in output.

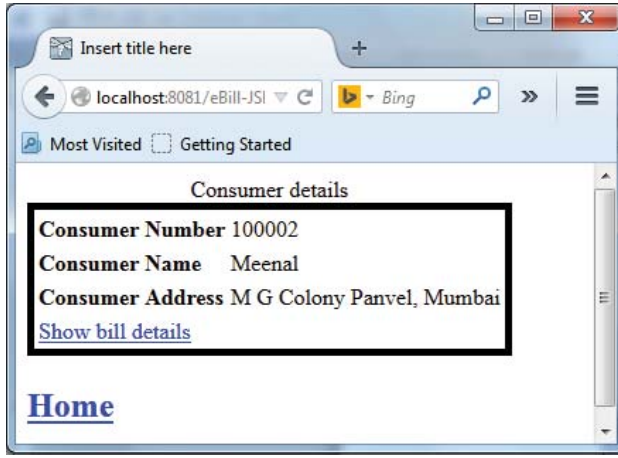


Figure 9:Show_Consumer.jsp

Spec 5: After clicking on the **Show bill details** link it will display the bill details for that consumer as below in the figure 10. The below page should be displayed when user clicks on the **Show bill details** link in the **Show_ConsumerList** page.

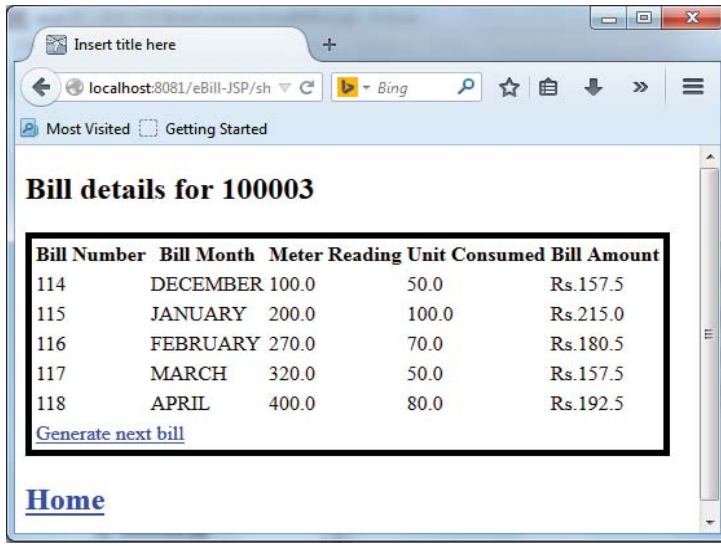


Figure 10:Show_Bills.jsp

Spec6: After clicking on **Generate nextbill** linkfollowing form as shown in figure 11 should be displayed to accept meter readings for that consumer.

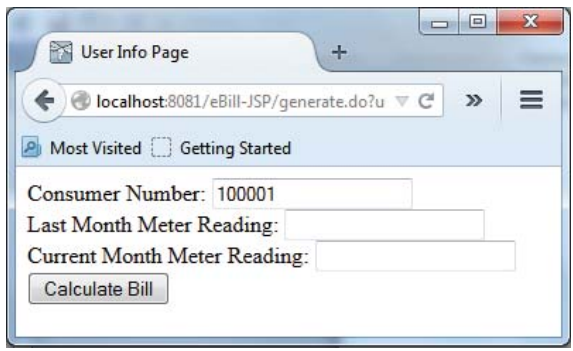


Figure 11:User_Info.jsp

Spec7: After clicking on the Calculate BillButton ,Bill must be generated as per the Servlet assignment 5.1 . Insert the details in the database and display the bill as shown in the following figure 12:

Figure 12:**Bill_Info.jsp**

Spec 8: In case of error, user defined error message should be displayed as shown in figure 13:

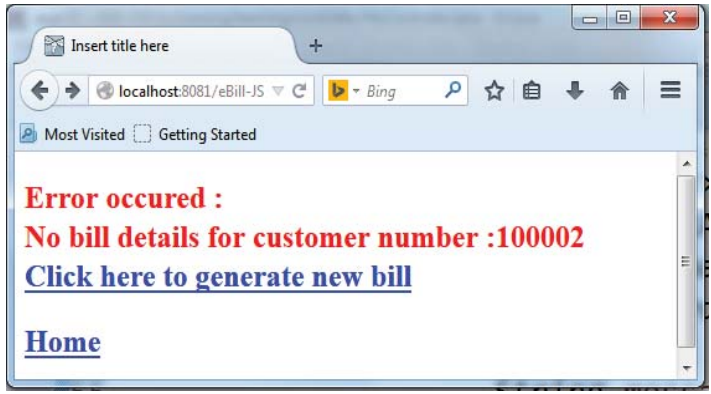


Figure 13:**Error.jsp**

2.3: JSP configuration using web.xml

Modify 2.1 Exercise with JSP configuration in web.xml.
Spec 1: Header.jsp created in 2.1 should be included to all JSP pages.
Spec 2: Scriptlet code should not be used to all JSP pages.
Spec 3: All JSP pages should evaluate the EL expressions.
Spec 4: Buffer size for all JSP pages should be 8 Kb.

Lab 3. **JSP Standard Tags Libraries (JSTL)**

Goals	• Using core Tag Libraries
Time	2hrs
Pre-requisite	XML Basics and Java

3.1: Using Core tag library : <c:set> and <c:out>

Write a JSP code that uses **c:set** tag to assign "**Hello World!**" value to the hello variable and then display the contents of the variable.

Start the WildFly Server to invoke **Hello_Core.jsp**. You must see the output as shown in figure 14:

Figure 14:**Output of Hello_Core.jsp using JSTL tags**

3.2: Using Core tag tag library: <c:if>

Create an html page that will accept username, age, and url. Check the age, if it lies above 18, then display a personalized welcome message to the user.
For example: Hello <username>, since you are <age> which is above 18 yrs, you will now be redirected to the url <url>.

Hint: use <c:if> and <c:url> and <c:redirect> tags for this.

3.3: Using Lambda Expression with EL

Modify 3.2 exercise with lambda expression with EL.

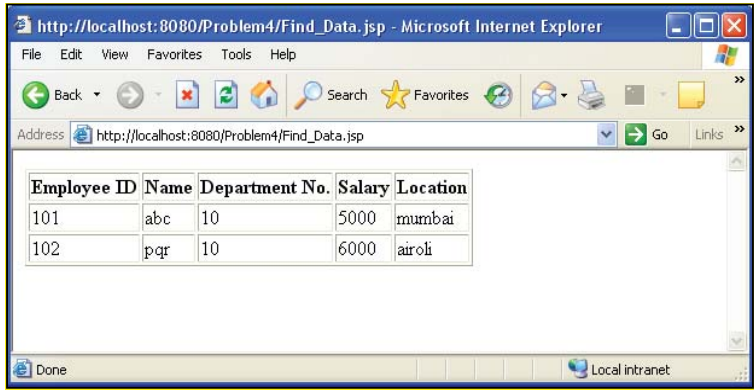
3.4 Modify**Electricity Bill** application which was developed in previous lab by developing JSP pages using appropriate JSTL tags and Lambda Expression, replacing all the scripting elements.

Appendix a: Extra Assignment

Assignment 1: JSP Directive

Develop a **Find_Info.jsp** page which accepts various inputs from user as shown in the figure 15 below. When user clicks the "**Find**" button, a **Find_Data.jsp** page display the result in a tabular format as shown in the figure 16. The employee details are stored in oracle database [Refer the table script]. Use page Directive appropriately. For the wrong inputs User defined error message should be displayed on Error.jsp page. Use MVC with Layered architecture given in the figure 17.

Figure 15:Find_Info.jsp



Employee ID	Name	Department No.	Salary	Location
101	abc	10	5000	mumbai
102	pqr	10	6000	airoli

Figure 16:Find_Info.jsp

The work flow of the application is shown in the figure given below:

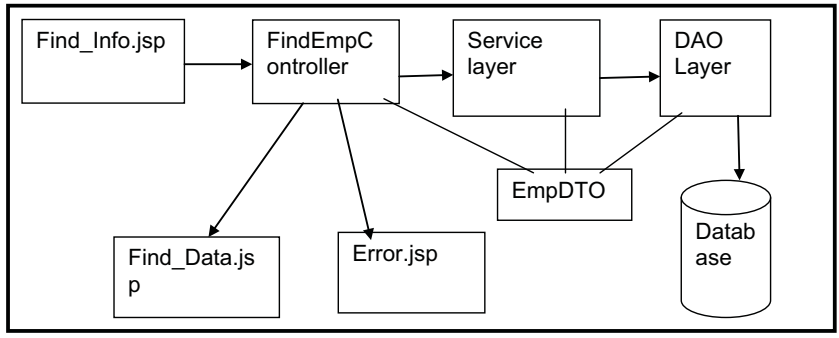


Figure 17: Architecture for application

Table Script :

```
CREATE TABLE employee_master(
        Emp_numNUBBER(4) PRIMARY KEY,
        Emp_nameVARCHAR2(25) NOT NULL,
        Dept_numNUMBER(2),
        Salary NUMBER(9,2),
```



```
        Location VARCHAR2(30)

);

INSERT INTO employee_masterVALUES(1001,'Smith',10, 5000,'Pune');
INSERT INTO employee_masterVALUES(1002,'Venkat',30, 7800,'Chennai');
INSERT INTO employee_masterVALUES(1003,'Meerah',20, 9000,'Surat');
INSERT INTO employee_masterVALUES(1004,'Saniya',30, 8000,'Noida');
INSERT INTO employee_masterVALUES(1005,'Shivam',30, 8500,'Noida');
INSERT INTO employee_masterVALUES(1006,'Kiran',10, 6000,'Pune');
COMMIT;
```

