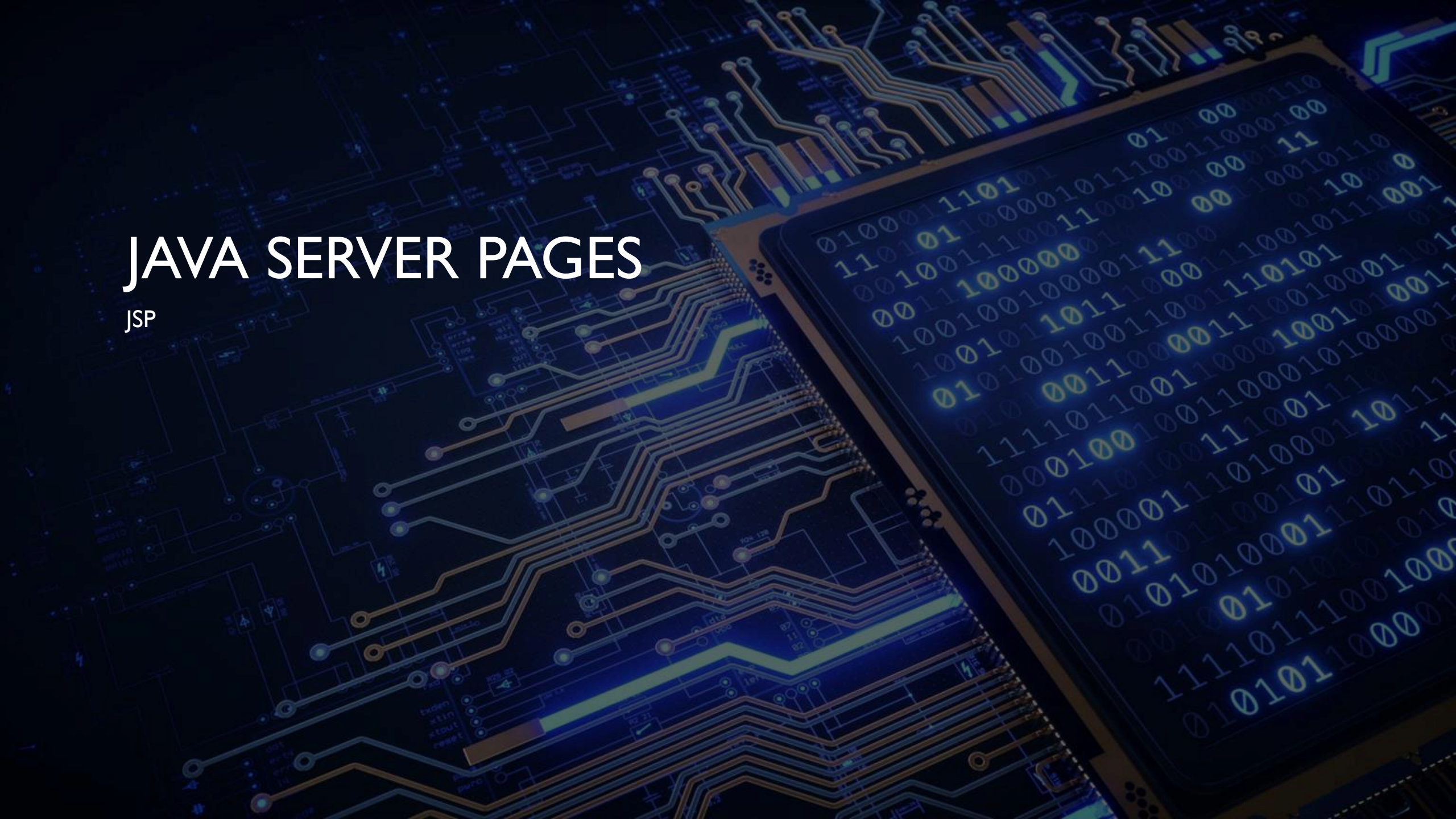
JAVA SERVER PAGES

JSP



OUTLINE

- JSP vs Servlets
- Basics of JSP
- JSP Life Cycle
- JSP Directives
- Using Java Beans
- JSP Expression Language
- Custom Tags
- JSTL
- Designing MVC Application

RESPONSIBILITIES OF SERVLETS

- Coding the presentation logic and business logic together is not a good practice
  - A change in any one of these requires the modification of the entire code
  - Programmers with different skill sets are required for creating and maintaining these
- Servlets, being Java programs, are best suited for coding business logic
- Servlets are not suitable to code presentation logic
  - It is not easy to mix static contents with dynamic contents in Servlets
  - As Servlets are not as easy as HTML, it will be difficult for web designers to use this technology
- A technology with the power of Servlets and ease of HTML is required to code presentation logic, so that web designers can also easily use it.

JAVA SERVER PAGES - BACKGROUND

- JSPs were invented to take care of the View.
- JSPs allow you to mix static (non-changing) HTML with snippets of Java code.
- You first generate the HTML page (giving it the .htm extension) and then, when you have decided what parts should have dynamic content, you add in the Java portions and rename the file .jsp

JAVA SERVER PAGES - BACKGROUND

- Converting a web page from .htm to .jsp is as simple as renaming the file.
- In the web application server, you can place a .jsp file anywhere you place an .htm file.
- You do not even need to compile a .jsp!
- No bothering with packages
- No bothering with CLASSPATHs.
- In fact, to use a JSP, all you need is a web server that is a web application server—meaning one that is configured to handle JSPs

SAMPLE JSP PAGE

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Index</title>
</head>
<body>
<h1>This is a JSP page</h1>
<%
    int i = 5;
    int j = 20;
    int sum = i + j;
    out.print("sum =" + sum);
%>
<h1>You have seen some java code above</h1>
</body>
</html>
```

Java code embedded inside HTML tags using **<%%>** tags. This is the basic structure of JSP.

JAVASERVER PAGES: BEHIND THE SCENES

- Outside of your view, the web application server is compiling your JSP.
- When a JSP is compiled, it is turned into a Servlet.
- In other words, a JSP is a Servlet in disguise.

WHAT HAPPENED TO THE INDEX JSP?

- The JSP which you had created would have been converted to a servlet file and compiled as class for servicing users request.
- You can find this file in the web server folder where the applications is deployed. The web server creates a temporary folder for extracting these files.

→ **NOTE:** The folder path varies between web servers.

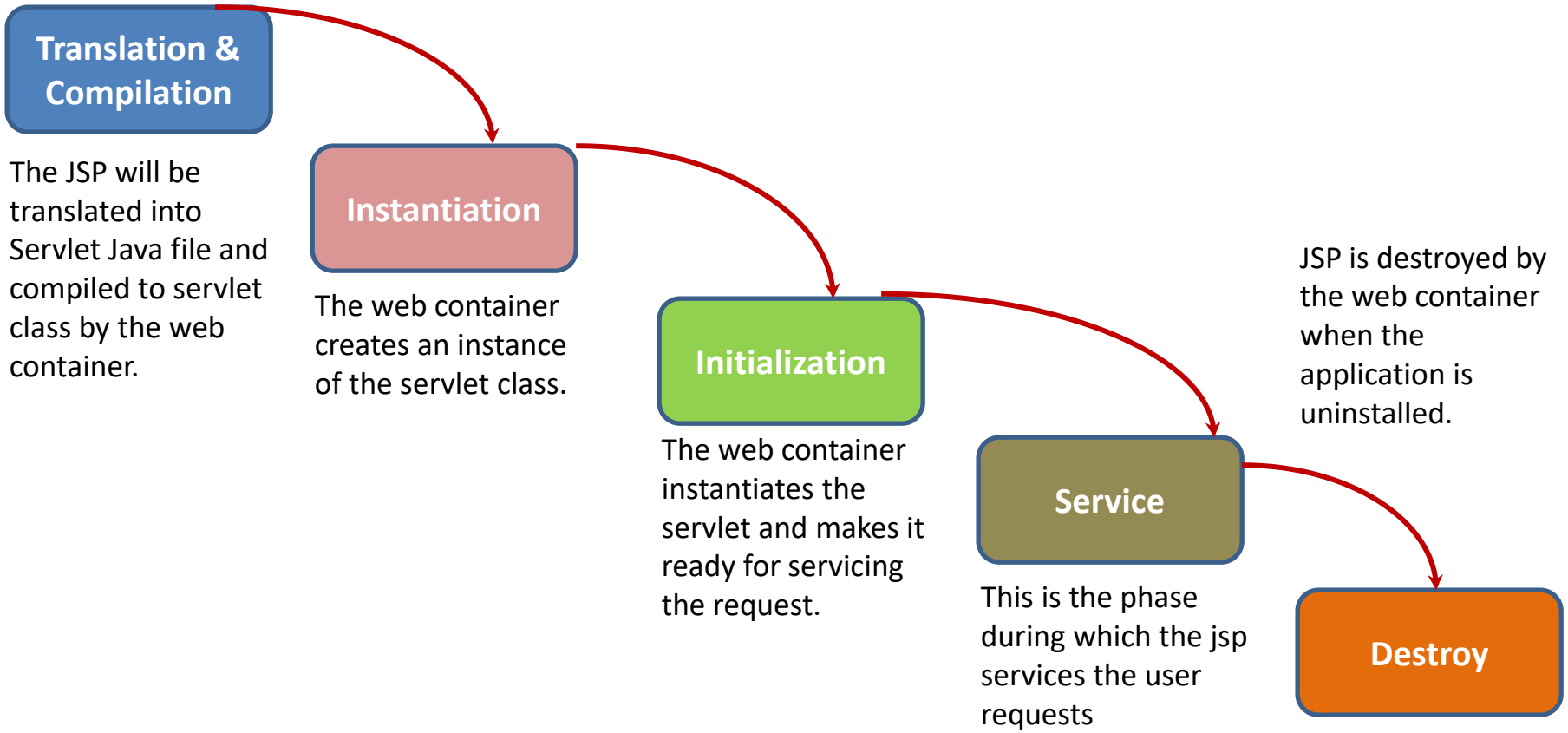
- **Lets see how our generated java file of index.jsp looks like.**

```
public final class index_jsp extends org.apache.jasper.runtime.HttpJspBase
    implements org.apache.jasper.runtime.JspSourceDependent {
  public Object getDependants() {
    return _jspx_dependants;
  }
  public void _jspInit() {
    _el_expressionfactory = _jspxFactory.getJspApplicationContext(getServletConfig().getServletContext())
                            .getExpressionFactory();
  }
  public void _jspDestroy() {
  }
  public void _jspService(HttpServletRequest request, HttpServletResponse response)
        throws java.io.IOException, ServletException {
     try {
      response.setContentType("text/html; charset=ISO-8859-1");
      pageContext = _jspxFactory.getPageContext(this, request, response,
                  null, true, 8192, true);
      _jspx_page_context = pageContext;
      out = pageContext.getOut();
      _jspx_out = out;
      out.write("<html>\r\n");
      out.write("<head>\r\n");
      out.write("<meta http-equiv=\"Content-Type\" content=\"text/html; charset=ISO-8859-1\">\r\n");
      out.write("<title>Index Page</title>\r\n");
      out.write("</head>\r\n");
      out.write("<body>\r\n");
      out.write("<h1 style=\"margin-left: 25%;\">First JSP Page</h1>\r\n");
      out.write("<h3>\r\n");
   out.print("Welcome to The world of JSP");
      out.write("\r\n");
      out.write("</h3>\r\n");
      out.write("<h1>You have successfully started JSP programming</h1>\r\n");
      out.write("</body>\r\n");
      out.write("</html>");
    } catch (Throwable t) {
      if (!(t instanceof SkipPageException)){
        out = _jspx_out;
        if (out != null && out.getBufferSize() != 0)
          try { out.clearBuffer(); } catch (java.io.IOException e) {}
        if (_jspx_page_context != null) _jspx_page_context.handlePageException(t);
      }
    } finally {
      _jspxFactory.releasePageContext(_jspx_page_context);
    }
  }
}
```

The service, init and destroy methods generated by the web container.

The Index.jsp translated to Java code.

JSP LIFE CYCLE PHASES



THE JSP LIFECYCLE

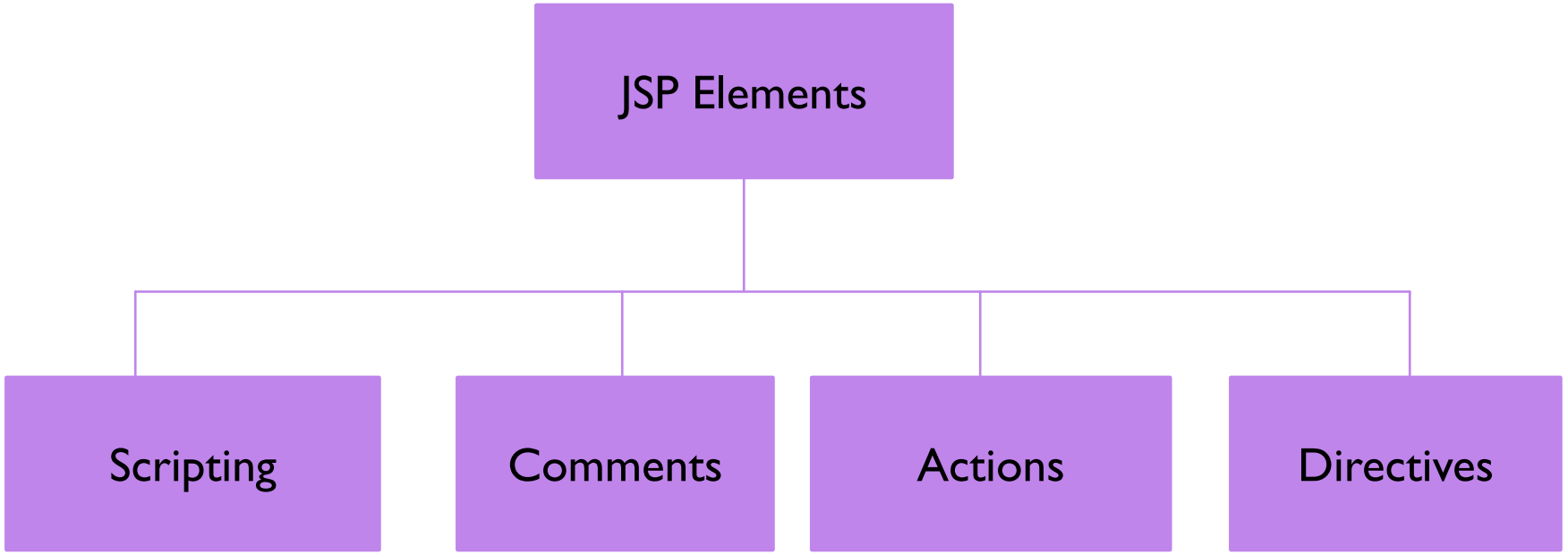
	Page first written	Request #1	Request #2	Server restarted	Request #3	Request #4	Page modified	Request #5	Request #6
JSP page translated into servlet		Yes	No		No	No		Yes	No
Servlet compiled		Yes	No		No	No		Yes	No
Servlet instantiated and loaded into server's memory		Yes	No		Yes	No		Yes	No
init (or equivalent) called		Yes	No		Yes	No		Yes	No
doGet (or equivalent) called		Yes	Yes		Yes	Yes		Yes	Yes

JSP LIFE CYCLE METHODS

- The following methods will be generated by the web container when translating the JSP to the Servlet Java file.
  - **jspInit()**
    - The web container calls the jspInit() to initialize the servlet instance generated. It is invoked before servicing the client request and invoke only once for a servlet instance.
  - **_jspservice()**
    - The container calls the jspservice() for each user request, passing it the request and the response objects.
  - **jspDestroy()**
    - The container calls this when it decides take the instance out of service. It is the last method called in the servlet instance.

JSP ELEMENTS

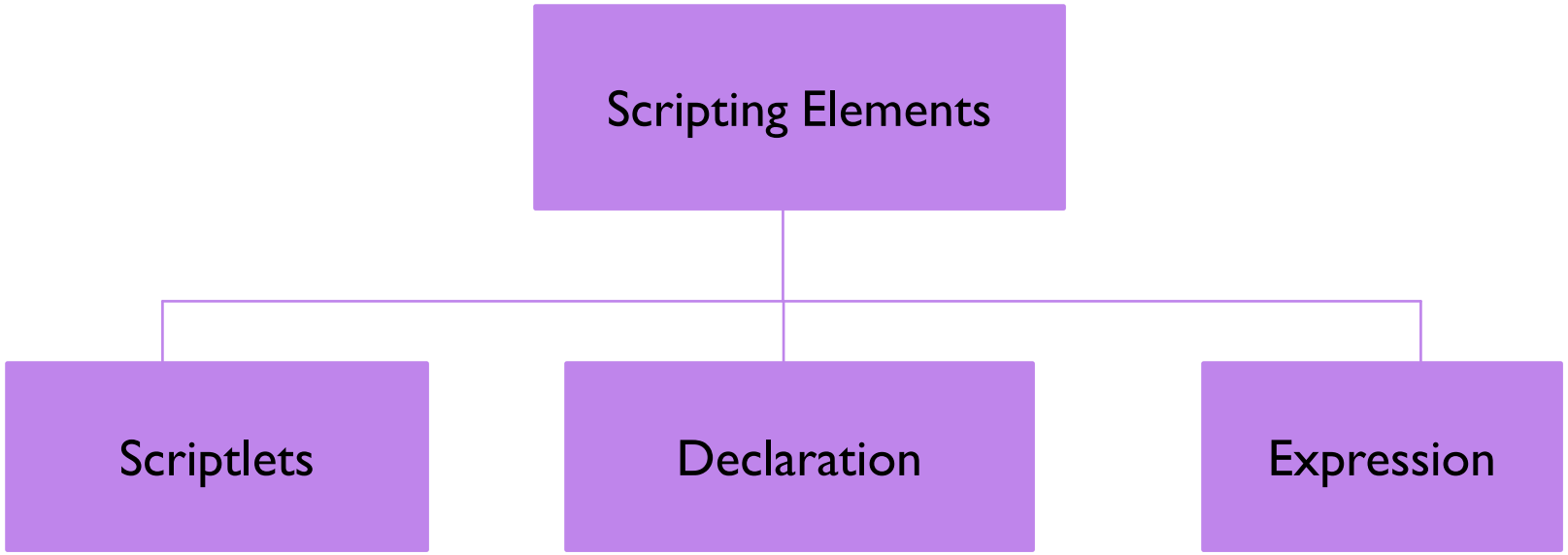
There are four types of elements in JSP.



SCRIPTING ELEMENTS

Scripting element are used to embed java code in JSP files.

There are three types of scripting elements,



SCRIPTLET ELEMENT

- Used to embed java code in JSP pages.
- Contents of JSP scriptlet goes into ***_jspService()*** method during the translation phase.
- Code should comply with syntactical and semantic construct of java.
- Embedded between **<% and %>** delimiters.

Syntax: <% Java code goes in here**%>**

Example: To print a variable value,

```
<%
      String username = “ABC" ;
      out.println ( username ) ;
 %>
```

DECLARATIONS ELEMENT

- Declarations are used to declare, define methods & instance variables.
- Declarations can also be used to override jspInit() and jspDestroy() methods
- Declaration tag does not produce any output that is sent to client.
- The methods and classes declared will be translated as class level variables and methods during translation.

HOW TO DECLARE?

- Methods or variables are declared using **<%!** and **%>** delimiters
- **Syntax:** <%! variable= 0; %>
- **Example:** This declares a variable count as int and set value 10.

```
<%! int count= 10; %>
```

EXPRESSION ELEMENT

- Used to write dynamic content back to the client browser.
- Used in place of ***out.print()*** method.
- Only expressions are supported inside the tag. Declarations of methods and variables is **not** possible inside this tag.
- During translation the return type of expression goes as argument into ***out.print()*** method.
- Expression should ***not*** be ended with a semicolon (**;**) since semicolon are automatically added during translation.

HOW TO USE EXPRESSIONS?

- Expressions are embedded in **<%=** and **%>** delimiters
- **Syntax: <%=** expression I **%>**
- **Example:** To print the date dynamically for each client request.

  <HTML> <BODY> Hello! The time is now **<%=** new java.util.Date() %> </BODY> </HTML>

- The date expression will be evaluated and the current date will be printed in the HTML rendered.

COMMENTS

- There are two type of comments supported by JSP

1. HTML comment

```
<!-- This is a comment --!>
```

2. JSP Comment

```
<%-- This is a comment --%>
```

- HTML comments are passed during the translation phase and hence *can be viewed* in the page source in the browser.
- JSP comments are converted to normal java comments during the translation process and *will not appear* in the output page source.

JSP IMPLICIT OBJECTS

- Implicit objects in JSP are the objects that are created by the web container automatically and the container makes them available to the JSP to access it.

- Implicit objects are available only inside the ***_jspService()*** method hence cannot be accessed anywhere outside.

IMPLICIT OBJECTS

- Some of the important objects and their classes are as follows
  - out (JSPWriter)
  - request (HttpServletRequest)
  - response (HttpServletResponse)
  - session (HttpSession)
  - config (ServletConfig)
  - exception (Throwable)
  - Application(ServletContext)
  - Page
  - pageContext
- These objects will be declared by the generated Servlet and hence the statements we write in JSP using these variables will get a meaning once they are pasted in the Servlet code

WHAT IS JSP ACTION ELEMENTS?

- JSP Action Elements are a set of predefined tags provided by the JSP container to perform some common tasks thus reducing the java code in JSP.

- Some of the common tasks are
  - Instantiate java bean object.
  - Setting values to beans.
  - Reading values from beans.
  - Forward the request to another resource.
  - Including another resource.

ACTION ELEMENTS IN JSP

- The following are the action tags available in JSP
  - jsp:include
  - jsp:forward
  - jsp:param
  - jsp:usebean
  - jsp:setProperty
  - jsp:getProperty
  - jsp:fallback
  - jsp:element
  - jsp:body
  - jsp:text
  - jsp:attribute
  - jsp:plugin

STANDARD ACTION ELEMENTS

- An Action Element consists of a Start Tag, a Body and an End Tag.

```
<jsp:forward page="nextPage.jsp">
  <jsp:param name="aParam" value="aValue" />
</jsp:forward>
```

The **<jsp:param>** action in the body of a **<jsp:forward>** is used to specify additional request parameters for the target resource.

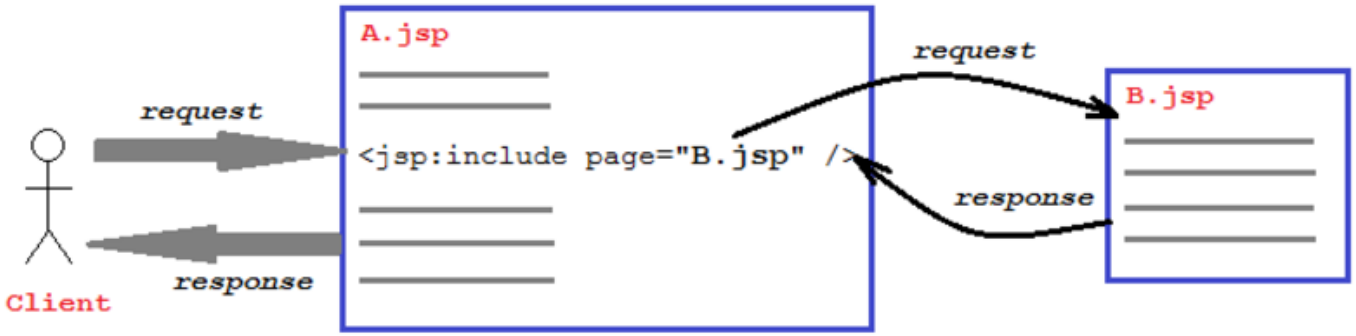
- If the Action Element does not have a body, you can use a shorthand notation:

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

- The <jsp:forward> action passes the request processing control to another JSP or Servlet in the same web application. Execution of the current page is terminated at that point.

JSP:INCLUDE

- Used for dynamically including the pages
- Includes the output of the included page during run time
- Contents of the page are not included – Only the response is included.



<jsp:include page="PageName" />

Example : Assume that the following line is included in ***index.jsp***

<jsp:include page="myPage.jsp" />

JSP DIRECTIVES

WHAT IS A JSP DIRECTIVE?

- A directive provides meta data information about the JSP file to the web container. Web container uses this during the translation/compilation phase of the JSP cycle

- Examples
  - Importing tag libraries
  - Import required classes
  - Set output buffering options
  - Include content from external files

TYPES OF DIRECTIVES

Directive	Purpose
Page	Provides information about page, such as scripting language that is used, content type, or buffer size etc.
Include	Used to include the content of external files.
Taglib	Used to import custom tags defined in tag libraries. Custom tags are typically developed by developers.

JSP DIRECTIVE SYNTAX

- A JSP directive is declared using the following syntax.

  <%@directive attribute="value" %>

  where

  - directive
    - The type of directive (page ,taglib or include)
  - attribute
    - Represents the behavior to be set for the directive to act upon.

PAGE DIRECTIVE

- The page directive is used to provide the metadata about the JSP page to the container.
- Page directives may be coded anywhere in JSP page.
- By standards, page directives are coded at the top of the JSP page.
- A page can have any number of page directives .
- Any attribute except the import attribute can be used only once in a JSP page.
- Single can contain more than one attribute specified.
- **Syntax :**

  <%@page attribute1 ="value" attribute2="value" %>

ATTRIBUTES FOR PAGE DIRECTIVE

Attribute	Purpose
buffer	Specifies a buffering model for the output stream. Same as the servlet buffer. <%@ page buffer="none|8kb|sizekb" %>
autoFlush	Controls the behavior of the servlet output buffer. <%@ page autoFlush="True\False" %>
contentType	Defines the character encoding scheme. <%@ page contentType="text/html;charset=ISO-8859-1" %>
errorPage	The errorPage directive takes a valid relative URL to a JSP file to which the control is redirected in case of any exceptions.. <%@ page errorPage="relativeURL" %>
isErrorPage	Works in tandem with the page errorPage directive and specifies that this JSP is an error page. <%@ page isErrorPage="true|false" %>

ATTRIBUTES FOR PAGE DIRECTIVE

Attribute	Purpose
buffer	Specifies a buffering model for the output stream. Same as the servlet buffer. <%@ page buffer="none|8kb|sizekb" %>
autoFlush	Controls the behavior of the servlet output buffer. <%@ page autoFlush="True\False" %>
contentType	Defines the character encoding scheme. <%@ page contentType="text/html;charset=ISO-8859-1" %>
errorPage	The errorPage directive takes a valid relative URL to a JSP file to which the control is redirected in case of any exceptions.. <%@ page errorPage="relativeURL" %>
isErrorPage	Works in tandem with the page errorPage directive and specifies that this JSP is an error page. <%@ page isErrorPage="true|false" %>

ATTRIBUTES FOR PAGE DIRECTIVE

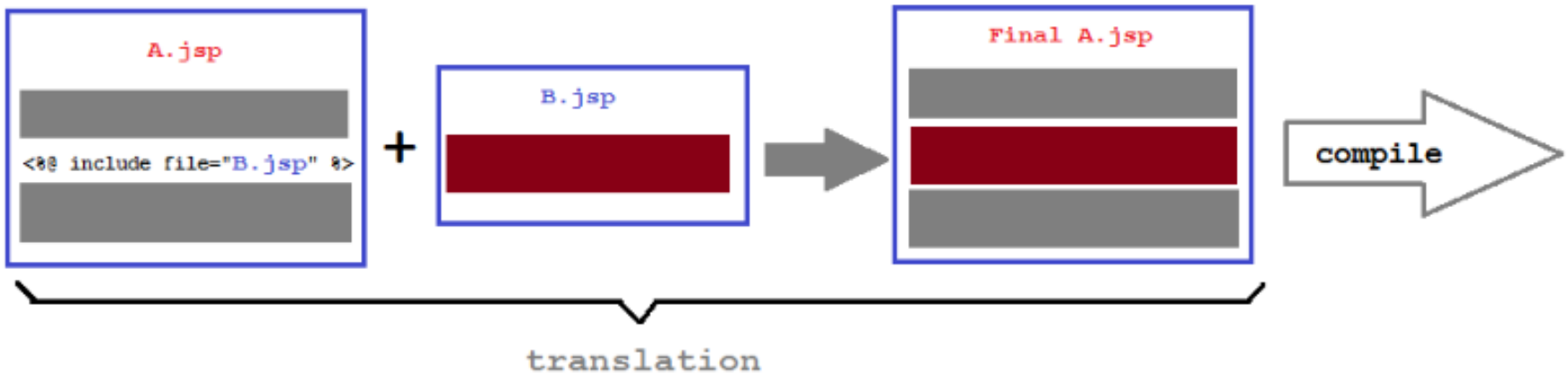
Attribute	Purpose
language	Defines the programming language used in the JSP page. <%@page language="java" %>
session	Specifies whether or not the JSP page participates in HTTP sessions <%@page language="java" session="true" %>
isELIgnored	Specifies whether or not EL expression within the JSP page will be ignored. <%@ page isELIgnored="True|False" %>
isScriptingEnabled	Determines if scripting elements are allowed for use. <%@ page isScriptingEnabled="True|False" %> Setting to false will throw error during translation if your page contains any scripting element such as scriplets , expression etc

COMMONLY USED ATTRIBUTES

- Following 4 attributes which are commonly used in application development
  - Buffer
  - Error Page
  - Import
  - Session

INCLUDE DIRECTIVE

This directive inserts a HTML file or a JSP file into another JSP file at translation time as illustrated below,



MORE ON INCLUDE DIRECTIVE

- The include process is static, it means that the text of the included file is added to the JSP file. (Similar to copy pasting the contents).
- The included file can be a JSP file, HTML file, or text file.
- If the included page is a JSP page it will be translated along with the main JSP page.
- **Note :**
  - Be careful that the included file ***should not*** contain ***<html>, </html>, <body>, or </body>*** tags.
  - Because the entire content of the included file is added to the main JSP file, these tags would conflict with the same tags in the main JSP file, causing an error.

HOW TO CREATE AN INCLUDE DIRECTIVE?

- Include directive is created using the following syntax

  <%@include attribute =“value”%>

- Include can have only ***file*** attribute.

  <%@include file =“value”%>

- Where file specifies the relative path of the file to be included.

  Example :

  <%@include file =“header.html”%>

  -> includes a file named header.html

JSP:INCLUDE VS. <%@ INCLUDE …>

	jsp:include	<%@include ... %>
Basic Syntax	<jsp:include page="…"/>	<%@include file="..." %>
When inclusion occurs	Request Time	Page Translation Time
What is included	Output of page	Contents of file
Number of resulting servlets	Two	One
Can included page set response headers that affect main page?	No	Yes
Can included page define fields or methods that main page uses?	No	Yes
Does main page need to be updated when included page changes?	No	Yes

WHICH SHOULD BE USED WHEN?

- Use jsp:include whenever possible
  - Changes to included page do not require any manual updates
  - Speed difference between jsp:include and the include directive (@include) is insignificant

- The include directive (<%@ include …%>) has additional power, however
  - Main page
    - <%! int accessCount = 0; %>

  - Included page
    - <%@ include file="snippet.jsp" %>
    - <%= accessCount++%>

USING JAVA BEAN

SERVER SIDE JAVA BEAN

- A server side java bean is a class used to store the details of real world entities

  Example: Employee -> Employee Name and Employee Salary,

  Student -> Student Name, Student Address.

- Bean is a plain java class which contains
  - **Fields (or) Properties:** Fields to store the data, **example:** Employee Name, Salary.
  - **Methods:** Methods for retrieving and modifying the attributes like setEmployeeName(), setStudentAddress(). The methods are referred to as accessors/mutator.

JAVA BEAN DESIGN CONVENTIONS

- Java classes that follow certain conventions
  - Must have a zero-argument (empty) constructor
    - You can satisfy this requirement either by explicitly defining such a constructor or by omitting all constructors
  - Should have no public instance variables (fields)
    - Use accessor methods instead of allowing direct access to fields
  - Persistent values should be accessed through method called get*Xxx* and set*Xxx*
  - If class has method getTitle that returns a String, class is said to have a String *property* named title.
  - Boolean properties use is*Xxx* instead of get*Xxx*

NEED FOR BEANS IN JSP

- Beans are used to JSP for collectively storing some information .
- Beans makes transfer of data between JSP's easier.
- For example if you are handling with a registration form all the registration details can be loaded into a RegistrationBean and can be transported across other components as a single object.
- Convenient correspondence between request parameters and object properties.

USING BEANS

- jsp:useBean
  - In the simplest case, this element builds a new bean. It is normally used as follows:

```
<jsp:useBean id="beanName" class="package.Class" />
```

- jsp:getProperty
  - This element reads and outputs the value of a bean property. It is used as follows:

```
<jsp:getProperty name="beanName" property="propertyName" />
```

- jsp:setProperty
  - This element modifies a bean property (i.e., calls a method of the form set*Xxx*). It is normally used as follows:

```
<jsp:setProperty name="beanName" property="propertyName" value="propertyValue" />
```

HOW JSP:USEBEAN WORKS?

- *Example :*

  *<jsp:useBean id=“*user" *class=“*com.catp.beans.UserBean*“ scope=“request “ />*

- How it works?
  - Attempts to locate a Bean with the name “UserBean” in the request scope.
  - If it finds the Bean, stores a reference in the variable user.
  - If it does not find the Bean, instantiates a bean using the class UserBean, and stores the reference to the variable user.

ASSOCIATING INDIVIDUAL PROPERTIES WITH INPUT PARAMETERS

- Use the param attribute of jsp:setProperty to indicate that
  - Value should come from specified request parameter
  - Simple automatic type conversion should be performed for properties that expect values of standard types.

```
<jsp:setProperty name="beanName" property="propertyName" param="Name of request Parameter" />
```

ASSOCIATING ALL PROPERTIES WITH INPUT PARAMETERS

- Use "*" for the value of the property attribute of jsp:setProperty to indicate that
  - Value should come from request parameter whose name matches property name
  - Simple automatic type conversion should be performed

- <jsp:useBean id="entry“ class="SaleEntry" />

- <jsp:setProperty name="entry" property="*" />
  - This is extremely convenient for making "form beans" – objects whose properties are filled in from a form submission.
  - You can even divide the process up across multiple forms, where each submission fills in part of the object.

THREE SMALL WARNINGS ARE IN ORDER

- No action is taken when an input parameter is missing.
- Automatic type conversion does not guard against illegal values as effectively as does manual type conversion.
- Bean property names and request parameters are case sensitive.

CONDITIONAL BEAN OPERATIONS

- Bean conditionally created
  - jsp:useBean results in new bean being instantiated only if no bean with same id and scope can be found.
  - If a bean with same id and scope is found, the preexisting bean is simply bound to variable referenced by id.
- Bean properties conditionally set
  - <jsp:useBean ... />

    replaced by

  - <jsp:useBean ...>statements</jsp:useBean>
    - The statements (jsp:setProperty elements) are executed *only* if a new bean is created, not if an existing bean is found.

BEAN OBJECT SCOPE

- Page: is available only within the JSP page and is destroyed when the page has finished generating its output for the request.
- request: valid for the current request and is destroyed when the response is sent
- session: valid for a user session and is destroyed when the session is destroyed
- application: valid throughout the application and is destroyed when the web application is destroyed/uninstalled.

JSP EXPRESSION LANGUAGE

EXPRESSION LANGUAGE

- JSP 2.0 introduced a shorthand language for evaluating and outputting the values of Java objects that are stored in standard locations.
- This **expression language (EL)** is one of the two most important new features of JSP 2.0.
- The other is the ability to define custom tags with JSP syntax instead of Java syntax.

MAIN POINT OF ALL OF EL IN ONE SLIDE (REALLY!)

- When using MVC in JSP 2.0-compliant server with current web.xml version, change:

  <jsp:useBean id="*someName*" type="*somePackage.someClass*" scope="*request, session, or application*"/>
  <jsp:getProperty name="someName" property="someProperty"/>

- To:

  ${someName.someProperty}

ADVANTAGES OF THE EXPRESSION LANGUAGE

- Concise access to stored objects.
  - To output a "scoped variable" named saleItem, use ${saleItem}.
  - Scoped variables are objects stored with setAttribute in the PageContext, HttpServletRequest, HttpSession, or ServletContext
- Shorthand notation for bean properties.
  - To output the companyName property of a scoped variable named company, use ${company.companyName}.
  - To access the firstName property of the president property of a scoped variable named company, use ${company.president.firstName}.
- Simple access to collection elements.
  - To access an element of an array, List, or Map, use ${variable[indexOrKey]}.
  - Provided that the index or key is in a form that is legal for Java variable names, the dot notation for beans is interchangeable with the bracket notation for collections.

ADVANTAGES OF THE EXPRESSION LANGUAGE(CONTD)

- Short and clear access to request parameters, cookies, and other request data.
- A small but useful set of simple operators.
- Conditional output.
  - To choose among output options, you do not have to resort to Java scripting elements. Instead, you can use ${test ? option1 : option2}.
- Automatic type conversion.
- Empty values instead of error messages.

ACTIVATING THE EXPRESSION LANGUAGE

- Available only in servers that support JSP 2.0 (servlets 2.4)
  - E.g., Tomcat 5, not Tomcat 4
  - For a full list of compliant servers, see http://www.theserverside.com/matrix
- You must use the JSP 2.0 web.xml file (Servlets 2.4)

INVOKING THE EXPRESSION LANGUAGE

- Basic form: ${expression}
  - These EL elements can appear in ordinary text or in JSP tag attributes, provided that those attributes permit regular JSP expressions. For example:
    - <UL>

      <LI>Name: ${expression1}

      <LI>Address: ${expression2}

      </UL>

    - <jsp:include page="${expression3}" />
- The EL in tag attributes
  - You can use multiple expressions (possibly intermixed with static text) and the results are coerced to strings and concatenated. For example:
    - <jsp:include page="${expr1}blah${expr2}" />

PREVENTING EXPRESSION LANGUAGE EVALUATION

- What if JSP page contains ${ ?
- Deactivating the EL in an entire Web application.
  - Use a web.xml file that refers to servlets 2.3 (JSP 1.2) or earlier.
- Deactivating the expression language in multiple JSP pages.
  - Use the jsp-property-group web.xml element
- Deactivating the expression language in individual JSP pages.
  - Use <%@ page isELIgnored="true" %>
    - This is particularly useful in pages that use JSTL
- Deactivating individual EL statements.
  - In JSP 1.2 pages that need to be ported unmodified across multiple JSP versions (with no web.xml changes), you can replace $ with $, the HTML character entity for $.
  - In JSP 2.0 pages that contain both EL statements and literal ${ strings, you can use \${ when you want ${ in the output

DEACTIVATING THE EL IN AN ENTIRE WEB APPLICATION

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

<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>


    </web-app>
```

DEACTIVATING THE EL IN MULTIPLE JSP PAGES

- In web.xml

```
<jsp-property-group>
    <url-pattern>/*.jsp</url-pattern>
    <el-ignored>true</el-ignored>
</jsp-property-group>
```

PREVENTING USE OF STANDARD SCRIPTING ELEMENTS

- To enforce EL-only with no scripting, use scripting-invalid in web.xml

```
<jsp-property-group>
        <url-pattern>*.jsp</url-pattern>
        <scripting-invalid>true</scripting-invalid>
</jsp-property-group>
```

ACCESSING SCOPED VARIABLES

- ${varName}
  - Means to search the PageContext, the HttpServletRequest, the HttpSession, and the ServletContext, *in that order*, and output the object with that attribute name.
  - PageContext does not apply with MVC.
- Equivalent forms
  - ${name}
  - <%= pageContext.findAttribute("name") %>
  - <jsp:useBean id="name" type="somePackage.SomeClass" scope="...">
  - <%= name %>

ACCESSING BEAN PROPERTIES

- ${varName.propertyName}
  - Means to find scoped variable of given name and output the specified bean property
- Equivalent forms
  - ${customer.firstName}
  - <%@ page import="bean.NameBean" %>
  - <% NameBean person = (NameBean)pageContext.findAttribute("customer"); %>
    <%= person.getFirstName() %>

ACCESSING BEAN PROPERTIES (CONTD.)

- Equivalent forms
  - ${customer.firstName}
  - <jsp:useBean id="customer“ type=“bean.NameBean"
    scope="*request, session, or application*" />
  - <jsp:getProperty name="customer“ property="firstName" />

- This is better than script on previous slide.
  - But, requires you to know the scope
  - And fails for subproperties.
  - No non-Java equivalent to ${customer.address.zipCode}

EQUIVALENCE OF DOT AND ARRAY NOTATIONS

- Equivalent forms
  - ${name.property}
  - ${name["property"]}

- Reasons for using array notation
  - To access arrays, lists, and other collections
  - To calculate the property name at request time.
    - {name1[name2]} (no quotes around name2)
  - To use names that are illegal as Java variable names
    - {foo["bar-baz"]}
    - {foo["bar.baz"]}

ACCESSING COLLECTIONS

- ${attributeName[entryName]}
- Works for
  - Array. Equivalent to
    - theArray[index]
  - List. Equivalent to
    - theList.get(index)
  - Map. Equivalent to
    - theMap.get(keyName)
- Equivalent forms (for HashMap)
  - ${stateCapitals["maryland"]}
  - ${stateCapitals.maryland}
  - But the following is illegal since 2 is not a legal var name
    - ${listVar.2}

REFERENCING IMPLICIT OBJECTS

- pageContext. The PageContext object.
  - E.g. ${pageContext.session.id}
- param and paramValues. Request params.
  - E.g. ${param.custID}
- header and headerValues. Request headers.
  - E.g. ${header.accept} or ${header["Accept"]}
  - ${header["Accept-Encoding"]}
- cookie. Cookie object (not cookie value).
  - E.g. ${cookie.userCookie.value} or ${cookie["userCookie"].value}
- initParam - Context initialization param.
- pageScope, requestScope, sessionScope, applicationScope.
  - Instead of searching scopes.

EL OPERATORS

- Arithmetic
  - + - * / div % mod
- Relational
  - == eq != ne < lt > gt <= le >= ge
- Logical
  - && and || or ! Not
- Empty
  - Empty
  - True for null, empty string, empty array, empty list, empty map. False otherwise.
- CAUTION
  - Use extremely sparingly to preserve MVC model

WHEN TO USE EL?

- EL should be used for accessing and presenting data from implicit objects.
- Don’t use EL operators and conditions for performing business logic which should be strictly done in business components.
- Business components: Here refers to the Java classes which is developed with business logic and are invoked by controllers (Typically servlets)

JSP CUSTOM TAGS

WHAT IS A CUSTOM TAG?

- Custom tags are similar to the JSP tags. The difference is these are user-defined tags.

- So what is a user defined Tag?
  - User defined tags are nothing but tags which are developed by programmers for performing specific functionalities.

  - Example: <CustomDate></ CustomDate>
    - Custom tag developed by programmers to print the current Date in a specified format.

  - The above tag when used in JSP will print the current system date.

WHY CUSTOM TAGS?

- Any common code which needs to be reused across the web application can be developed using custom tags.
- Easy to maintain as the logic is centralized.
- Any change to the logic just requires a one place change thus reducing the effort to change it.
  - Example: In the previous example if the date format changes only the custom tag "CustomDate" needs to be changed which gets automatically reflected throught the application.

TYPES OF TAGS

- Simple Tags :
  - A simple tag contains no body and no attributes.
  - Example: <tt:CustomDate/>

- Tags With Attributes :
  - A custom tag can have attributes.
  - Attributes are listed in the start tag and have the syntax: attribute name ="value".
  - Attribute are like configuration details for the custom tag
  - <tt:CustomDate attribute="value"/>

- Tags with Bodies :
  - A custom tag can contain custom, core tags, scripting elements and HTML text content between the start and end tag.

  - <tt:mytag>
    <h1> This is body inside the tag </h1>
    </tt:mytag>

HOW TO CREATE CUSTOM TAGS?

- The following steps are to be taken to create a custom tag.

1. Create a tag handler class with the custom logic.
2. Create a tag library descriptor (TLD) and configure the custom handler.
3. Import the custom tag library in JSP using taglib directive.
4. Start using the custom tag in a JSP page.

STEP 1: CREATE TAG HANDLER

- Tag handler is a Java class that holds the logic of the custom tag. This is triggered by the web container whenever it encounters the custom tag in a JSP file.
- Custom tag must implement or extend any of the following interfaces or classes
  1. Tag Interface
  2. BodyTag interface
  3. TagSupport class
  4. BodyTagSupport class
  5. SimpleTagSupport class

SIMPLETAGSUPPORT CLASS

- The *SimpleTagSupport* class is the base class intended to be used for developing tag handlers.

- The *SimpleTagSupport* class implements the *SimpleTag* interface and defines API's which can be overridden by the custom tag handlers .

METHODS IN SIMPLETAGSUPPORT

Method Name	Description
doTag()	The logic of the custom tag goes inside this method.
getParent()	Returns the parent of this tag.
setParent(JspTag tag)	Sets the parent of this tag.
setJspContext(JspContext context)	Stores the provided JSP context in the private jspContext field.
getJspContext()	Returns the page context passed in by the container via setJspContext.

STEP 2:

JSTL

WHAT IS JSTL?

- The **J**ava Server Pages **S**tandard **T**ag **L**ibrary (JSTL) is a collection of useful JSP tags which encapsulates some useful functionality widely used in many JSP applications.
- JSTL has some common functionalities implemented such as iteration, conditionals statements, tags for manipulating XML documents, internationalization tags, and SQL tags.

TYPES OF JSTL TAGS

- Based on the functionality there are four categories of JSTL tags
  1. Core Tags
  2. Formatting tags
  3. SQL tags
  4. XML tags

INSTALLING JSTL

- JSTL is distributed as a set of jar files that you simply dropped into your servlet container's classpath.
- You must use a JSP 2.0-compliant servlet container in order to use JSTL 1.1.
- Download a JSTL implementation from the Jakarta Tab Libs project. The binary distribution comes packaged as a .zip or .tar.gz.
- Copy all of the jar files in jakarta-taglibs/standard-1.0/lib to the /WEB-INF/lib directory of your Web application. This will include both JSTL-specific jars and all the jars that they depend on.
- Finally, import into your JSP page each JSTL library that the page will reference. This is done by adding the appropriate taglib directives at the top of your JSP page. The directives are:
  - **core**: <%@ taglib prefix="c" uri="http://java.sun.com/jstl/core" %>
  - **xml**: <%@ taglib prefix="x" uri="http://java.sun.com/jstl/xml" %>
  - **fmt**: <%@ taglib prefix="fmt" uri="http://java.sun.com/jstl/fmt" %>
  - **sql**: <%@ taglib prefix="sql" uri="http://java.sun.com/jstl/sql" %>

EXAMPLE-JSTL

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

<c:set var="hello" value="HelloWorld!"/>

<c:out value="${hello}"/>
```

DESIGNING MVC ARCHITECTURE

STRATEGIES FOR INVOKING DYNAMIC CODE FROM JSP.

Simple Application OR

Small Development Team

Complex Application OR

Large Development Team

- **Call Java code directly.** Place all Java code in JSP page. Appropriate only for very small amounts of code.
- **Call Java code indirectly.** Develop separate utility classes. Insert into JSP page only the Java code needed to invoke the utility classes.
- **Use beans.** Develop separate utility classes structured as beans. Use jsp:useBean, jsp:getProperty, and jsp:setProperty to invoke the code.
- **Use the MVC architecture.** Have a servlet respond to original request, look up data, and store results in beans. Forward to a JSP page to present results. JSP page uses beans.
- **Use the JSP expression language.** Use shorthand syntax to access and output object properties. Usually used in conjunction with beans and MVC.
- **Use custom tags.** Develop tag handler classes. Invoke the tag handlers with XML-like custom tags.

SERVLETS AND JSP: POSSIBILITIES FOR HANDLING A SINGLE REQUEST

- Servlet only. Works well when:
  - Output is a binary type. E.g.: an image
  - There is *no* output. E.g.: you are doing forwarding or redirection as in Search Engine example.
  - Format/layout of page is highly variable. E.g.: portal.
- JSP only. Works well when:
  - Output is mostly character data. E.g.: HTML
  - Format/layout mostly fixed.
- Combination (MVC architecture). Needed when:
  - A single request will result in multiple substantially different looking results.
  - You have a large development team with different team members doing the Web development and the business logic.
  - You perform complicated data processing, but have a relatively fixed layout.

IMPLEMENTING MVC WITH REQUESTDISPATCHER

1. Define beans to represent the data

2. Use a servlet to handle requests
   - Servlet reads request parameters, checks for missing and malformed data, etc.

3. Populate the beans
   - The servlet invokes business logic (application-specific code) or data-access code to obtain the results. Results are placed in the beans that were defined in step 1.

4. Store the bean in the request, session, or servlet context
   - The servlet calls setAttribute on the request, session, or servlet context objects to store a reference to the beans that represent the results of the request.

IMPLEMENTING MVC WITH REQUESTDISPATCHER (CONTD.)

5. Forward the request to a JSP page.
   - The servlet determines which JSP page is appropriate to the situation and uses the forward method of RequestDispatcher to transfer control to that page.

6. Extract the data from the beans.
   - The JSP page accesses beans with jsp:useBean and a scope matching the location of step 4. The page then uses jsp:getProperty to output the bean properties.
   - The JSP page can use Custom tag or EL expression also to output bean properties.
   - The JSP page does not create or modify the bean; it merely extracts and displays data that the servlet created.