Web Component Development Using Java

Session: 13

JSP Custom Tags

Objectives

- Explain JSP custom tags in JSP
- Describe the common terminology used in JSP custom tags
- Explain the working of custom tag libraries
- Explain the different types of custom tags available in JSP
- Explain how to create classic custom tags
- Explain use of Tag Extension API
- Explain Simple Tags API

JSP Custom Tags

- Custom tag allows Java developers to embed Java code in JSP pages.
- Using custom tags in a JSP page involves three steps:

Creating a Tag Library Descriptor (TLD)

- TLD file contains the information on each tag available in the library.
- It is an XML document.
- It used to validate the tags.

Creating a tag handler class

- Tag handler is a Java class that defines a tag described in the TLD file.

Creating a JSP page that will access the custom tags

- JSP page will use the tags defined in a tag library by using a taglib directive in the page before any custom tag is used.

Custom Tags Terminology 1-4

- **Tag library descriptor**
  - Is a XML file, which describes custom tags in a JSP program.
  - Contains the definitions of a custom tag and is saved with an `extension.tld`.
  - Imported to the JSP program by using the `<%@taglib>` directive.
- The JSP container creates an instance of the imported library descriptor file and traces the handler class of the custom tag.

Custom Tags Terminology 2-4

- The code snippet shows a sample tag library descriptor that describes a tag called `Name`.

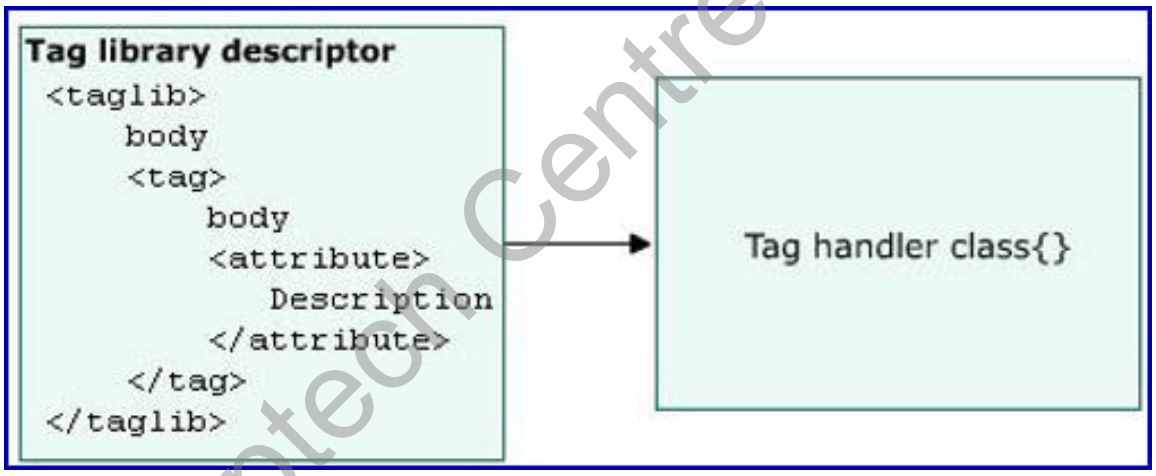
```
<taglib version="2.0" xmlns="http://java.sun.com/xml/
j2ee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee/web-
jsptaglibrary_2_0.xsd">

<!-- The tag library and the tag is described inside this
tag-->

<taglib>

<!-- The version of the tag library -->
    <tlib-version>2.0</tlib-version>

<!-- The JSP specification version for the tag library --
>
    <jsp-version>2.0</jsp-version>

<!-- This is the name assigned to the tag library -->
    <short-name>tags</short-name>

<!-- This specifies the path of the TLD file-->
    <uri>tag lib version id</uri>
```

Custom Tags Terminology 3-4

```
<!-- Provides a short description of the tag library-->
    <description>The tag library </description>

<!-- Provides a detailed description of the tag-->
<tag>

<!-- This is the name of the tag used in the JSP page-->
    <name>name</name>

<!-- This is the name of the tag handler class for the
concerned tag-->
    <tag-class>tags.NameTag</tag-class>

<!-- This specifies the type of body content. It can be empty,
JSP or tag-dependent -->
     <body-content>empty</body-content>

<!-- This describes the attribute passed with the tag-->
      <attribute>

<!-- This specifies the name of the attribute passed with the
tag- ->
        <name>name</name>
    </attribute>
</tag>
</taglib>
```

Custom Tags Terminology 4-4

- **Tag handler:**
  - It is a simple Java class file that contains the code for the functionality of the custom tag in the JSP program.
  - These are of two types, classic and simple.
  - Figure depicts the custom tags terminology.



Tag Libraries 1-2

- It is a collection of custom tags.
- It allows the developer to write reusable code fragments, which assign functionality to tags.
- When a tag is used in a JSP page, the library containing the tag has to be imported into the JSP page using the `<@%taglib>` directive.
- **Syntax:**

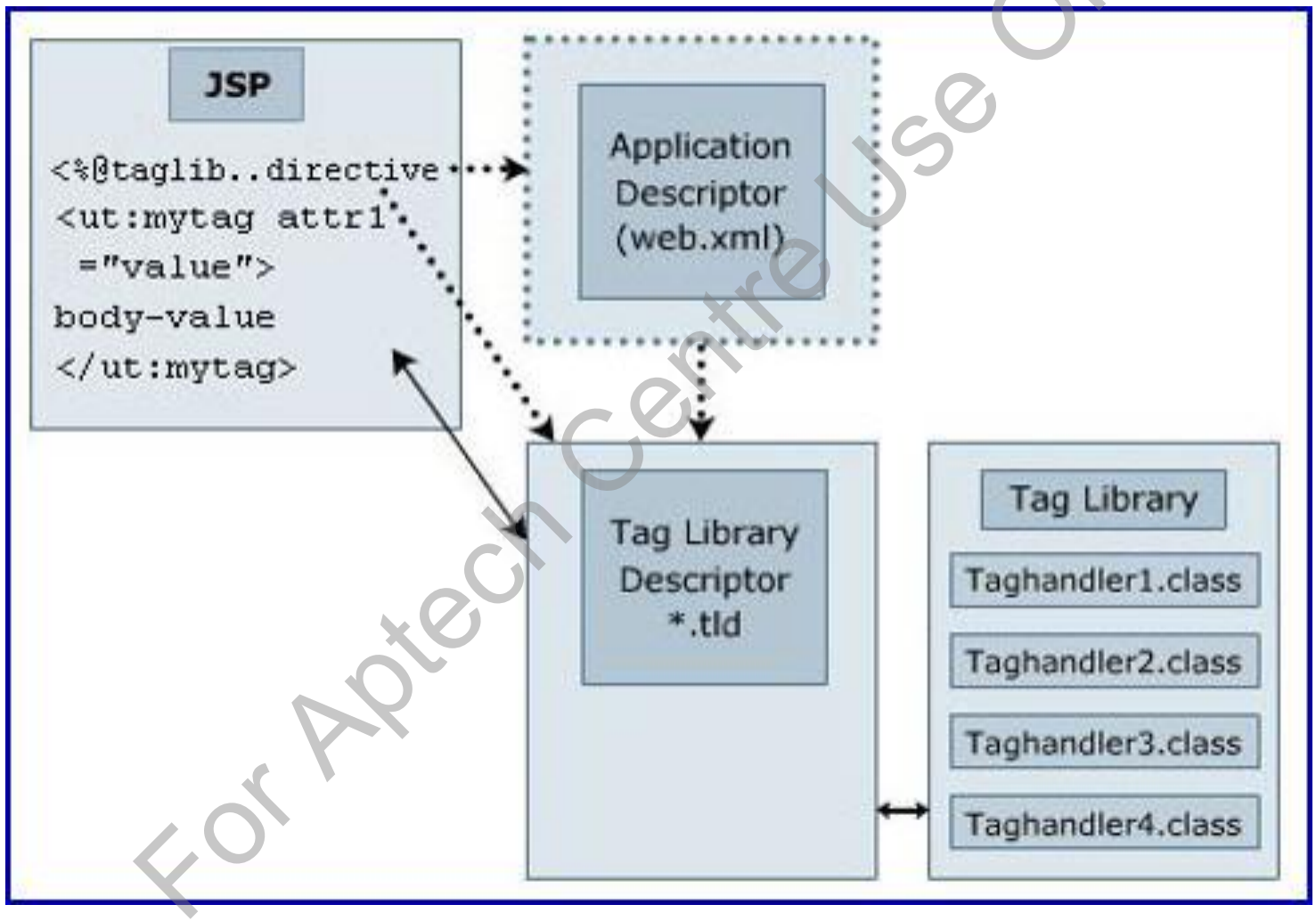
```
<%@taglib prefix="u" uri="/WEB-INF/ tlds/
sampleLib.tld" %>
```

where,

- `uri` is the path that uniquely identifies the TLD file of the tag
- `prefix` is the prefix attribute that distinguishes various tags in the JSP page

Tag Libraries 2-2

- Figure depicts the working of custom tag libraries.



Custom Tag Library

- The JSP engine is a component of Web container, which converts JSP codes to servlet codes.
- Jasper2 engine is the name of the JSP engine of Apache since Tomcat version 5.0.
- The functions of Jasper2 are as follows:

Analyze the JSP file and compile them to servlet code.

Allow a Java object instantiated for the JSP tag to be pooled and reused any time from memory.

Locating TLD File

- TLD file can reside in either any directory of a Web application such as:
  - `../WEB-INF/sampleLib.tld`
  - `../WEB-INF/tld/sampleLib.tld`
- They can be package in a jar file and place that jar in `../WEB-INF/lib/` directory along with related jars and any related resources.

Associating URIs with TLD File Locations

- A specific URI for each `.tld` file refers to its tag in a JSP page.
- This is done by assigning path to the `.tld` file to `uri` attribute.
- The `uri` attribute is mapped to the `.tld` file in two ways namely, implicit and explicit mapping.
- In implicit mapping the container reads all the `.tld` files present in the jar.
- Then, the JSP container automatically creates a mapping between the URI specifying the JAR location and the TLD file present inside the JAR file.
- **Syntax:**

```
<%@ taglib prefix= "u" uri="/WEB-
INF/TagJARfile.jar" %>
```

Explicit Mapping

- Is a method to map the URI to a TLD file.
- Defines reference to the tag library descriptor by the `<taglib>` element in the deployment descriptor, `web.xml`, of the Web application.
- Each `<taglib>` element contains two sub elements as follows:

<taglib-uri>	• This is the URI identifying a tag library.
<taglib-location>	• It describes the path to the tag library descriptor file for the custom tag.

- The code snippet uses the tag `<taglib-location>` element to indicate relative path of `AttributeTag.tld`.

```
<taglib>

        <taglib-uri>AttributeTag</taglib-uri>
        <taglib-location>/WEB-
        INF/tlds/AttributeTag.tld</taglib-location>

</taglib>
```

Resolving URIs to TLD File Locations

- In explicit mapping, the association of the URIs to the TLD files is called the resolution of URIs.
- This association is necessary to locate the tag library descriptor file, which defines the tag.
- If the value of the `uri` attribute matches any of the `<taglib-uri>` entries, the engine uses the value of the corresponding `<taglib-location>` to locate the actual TLD file.

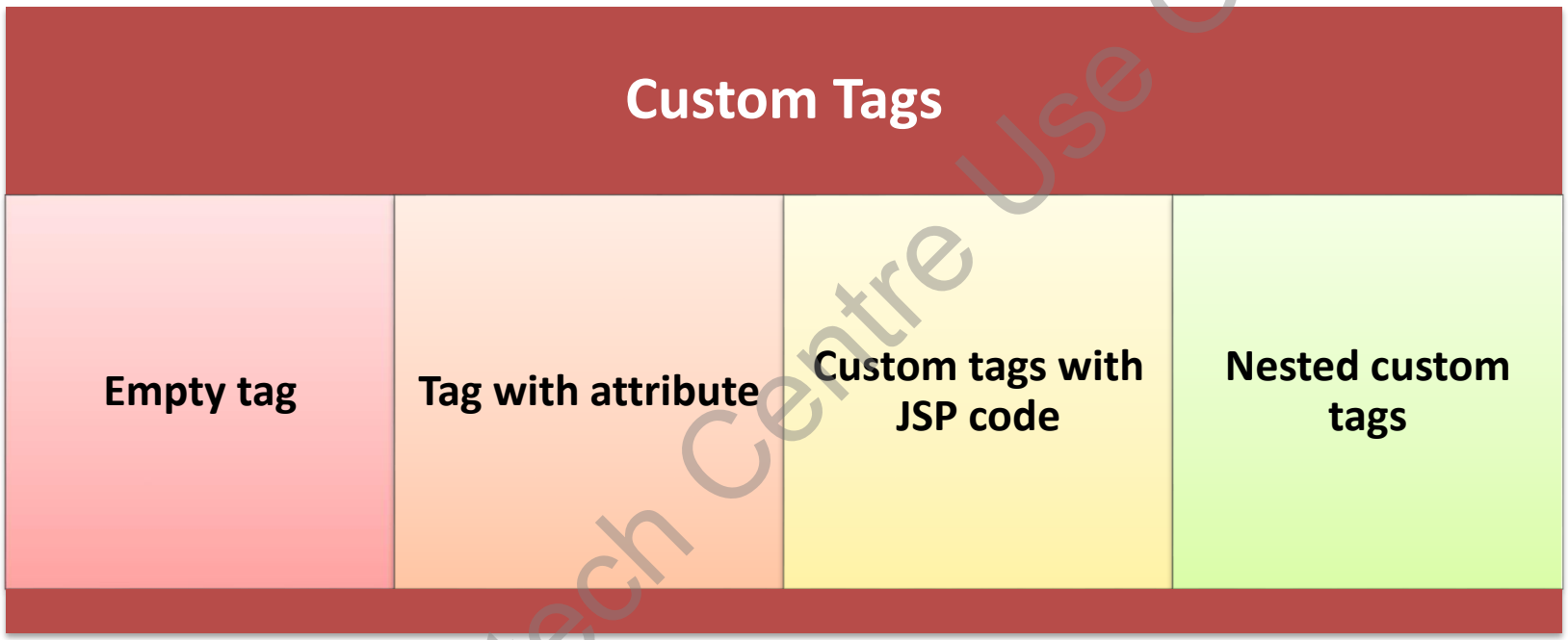
Tag Library Prefix

- To distinguish custom tags from one another, they are named uniquely by using prefix attributes in the `<%@taglib>` directive.
- The prefix of a tag is also used as a reference by the container to invoke its respective TLD file and tag handler class.
- **Syntax:**

```
<%@taglib prefix="ui" uri="/WEB-INF/tlds/sampleLib_ui.tld"
%>

<%@taglib prefix="logic" uri="/WEB-INF/tlds/sampleLib_logic.
tld" %>

<%@taglib prefix="data" uri="/WEB-
INF/tlds/sampleLib_data.tld" %>
```

Using Custom Tags in JSP Pages

- Following are the types of custom tags:



Tags with Attributes

- The custom tags with the attributes are called as **parameterized tags**.
- Attributes are passed to the tags as arguments are passed to a method.
- The code snippet customizes the behavior of a custom tag.

```
<html>

<body>

<%@ taglib prefix="tagPrefix" uri="sampleLib.tld"
%>

<h1>
<tagPrefix:TagsWithAttributesuserName="userName1"
/></h1>

</body>

</html>
```

Custom Tags with JSP Code

- Custom tags can contain JSP code as content inside them.
- The JSP code is simply evaluated or manipulated before being evaluated.
- The code snippet shows that the manipulation can be of converting the body content to a string and then formatting it as per requirements.

```
<html><body>
<%@ taglib prefix="tagPrefix" uri="sampleLib.tld" %>
   <tagPrefix:if condition="true">
   UserName is: <%= request.getParameter("userName") %>
   </tagPrefix:if>
</body></html>
```

Nested Custom Tags

- These are tags declared inside the tags.
- This is analogous to the `<table>` tag in an html page, which has `<tr>` and `<td>` tags inside it.
- One custom tag can made a container for another custom tag.
- The container is called the parent of the tags inside it.
- The code snippet shows the nested custom tags.

```
<html><body>

<%@ taglib prefix="tagPrefix" uri="sampleLib.tld" %>

<tagPrefix:switch conditionValue='<%= request.
getParameter("userName")%>'>

  <tagPrefix:case caseValue=" userName1">
       First User
  </tagPrefix:case>

<tagPrefix:case caseValue=" userName2" >
       Second User
</tagPrefix:case>
</tagPrefix:switch>

</body></html>
```

Classic Custom Tags

- The classic custom tags help in creating customized tags for JSP pages.
- It uses a tag handler class, which implements `Tag`, `IterationTag`, and `BodyTag` interfaces or extends the classes `TagSupport` or `BodyTagSupport`.
- It is then described in the Tag Library Descriptor (TLD) file.

<taglib> Element 1-2

- The `<taglib>` element is the top-level or root element of the tag library descriptor.
- This tag contains some sub tags inside it. They are as follows:

<tlib-version>	• Describes the version of the tag library implementation
<jsp-version>	• Defines the JSP version
<short-name>	• Uniquely identifies the tag library from the JSP page
<uri>	• Used as a unique resource identifier for the location of a tag library
<tag>	• Provides all the information required for a particular classic custom tag used in a JSP page

<taglib> Element 2-2

- The code snippet illustrates the use of the `<taglib>` elements along with its sub elements.

```
<taglib version="2.0"
xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2e
e web-jsptaglibrary_2_0.xsd">


   <tlib-version>1.0</tlib-version>
   <jsp-version>1.2</jsp-version>
   <short-name>attributetag</short-name>
   <uri>/WEB-INF/tlds/AttributeTag</uri>


</taglib>
```

<tag> Element 1-2

- The <tag> element contains the following sub elements to describe the tag characteristics:

<name>	• Assigns a unique name to the classic custom tag inside a JSP page.
<tag-class>	• Contains the name of the tag handler class that describes the functionality of a particular classic custom tag.
<body-content>	• Contains the content type for the body of the tag • This tag can be passed values such as empty, JSP, and Body-Content.
<attribute>	• Describes the attribute passed in the tag handler class to the JSP page containing the classic custom tag.

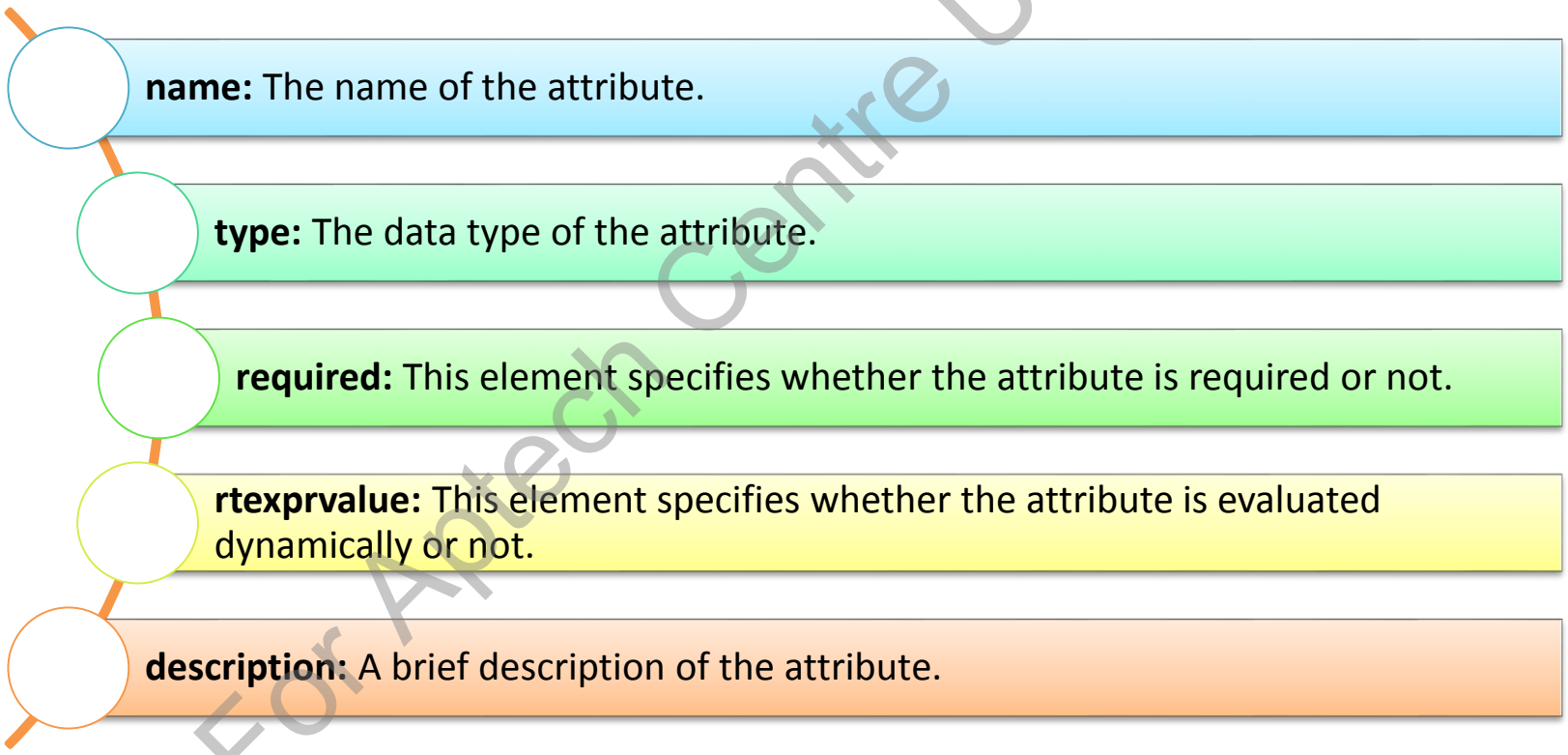
<tag> Element 2-2

- The code snippet illustrates the use of the <tag> elements along with its sub elements.

```
<tag>

<name>AttributeTag</name>
<tag-class>AttributeTag</tag-class>
<body-content>empty</body-content>

</tag>
```

<attribute> Element 1-2

- The `<attribute>` element inside the `<tag>` element describes the attribute passed in the tag handler class to the JSP page through the classic custom tag.
- It contains five sub tags:



<attribute> Element 2-2

- The code snippet demonstrates the `attribute` element.

```
<tag>
<name>MyTag</name>
<tag-class>pkg.MyTag</tag-class>
<body-content>JSP</body-content>

<attribute>
   <name>attr1</name>
   <required>true</required>
</attribute>

<attribute>
   <name>attr2</name>
   <required>false</required>
   <rtexprvalue>true</rtexprvalue>
</attribute>

</tag>
```

<body-content> Element 1-2

- It is defined within the `<tag>` element.
- It indicates whether the classic custom tag contains any content inside it or not.
- The body content can have one of three values as follows: **empty**, **JSP**, or **tagdependent**.

empty

- The code snippet shows the tag that does not contain any body inside it to process, then it is called an empty tag.

```
<tag>
    <name>AttributeTag</name>
    <tag-class>AttributeTag</tag-class>
    <body-content>empty</body-content>

<attribute>
</attribute>
</tag>
```

<body-content> Element 2-2

JSP

- The code snippet shows the JSP body content includes any other custom tag, core tag, scripting elements, or html text inside it.

```
<tag>
<name>UpperCase</name>
<tag-class>ucase.UpperCase</tag-class>
<body-content>JSP</body-content>
</tag>
```

tagdependent

- The code snippet specifies that the tag has a body, but its content is not to be interpreted by the JSP engine.

```
<tag>
<name>Query</name>
<tag-class>table.Query</tag-class>
<body-content>tagdependent</body-content>
</tag>
```

Tag Extension API

- It adds the tag functionalities to the language.
- It is part of the `javax.servlet.jsp.tagext` package.
- This contains a number of interfaces and classes, which have their exclusive methods.

Interfaces

- The three important interfaces defined in the tag extension API are as follows:

Tag

- It allows communication between a tag handler and the servlet of a JSP page.
- It is useful when a classic custom tag is without any body or even if it has body, then it is not for manipulation.

IterationTag

- It is used by tag handlers that require executing the tag, without manipulating its content.
- It extends to the `Tag` interface.

BodyTag

- It extends `IterationTag` and adds two methods for supporting the buffering of body contents: `doInitBody()` and `setBodyContent()`.

Classes

- The `tagext` package includes two important classes `TagSupport` and `BodyTagSupport` for implementing tag functionalities in a JSP page.

`TagSupport`

- It acts as the base class for most tag handlers which supports empty tag, tag with attributes, and a tag with body iterations.
- It implements the `Tag` and `IterationTag` interfaces.
- It contains methods such as `doStartTag()`, `doEndTag()`, and `doAfterBody()`.

`BodyTagSupport`

- It support tags that need to access and manipulate the body content of a tag.
- It implements the `BodyTag` interface and extends the `TagSupport` class.
- It contains the following methods: `setBodyContent()` and `doInitBody()`.

Exceptions

- The tag handler classes use the exception classes which are defined in the `javax.servlet.jsp` package.

JspException

- It is thrown by a tag handler to indicate some unrecoverable error while processing a JSP page.

JspTagException

- It is a sub class of the `JspException`, which is thrown when an error is encountered inside the tag handler class while processing any tag.

Methods of Tag Interface 1-3

- **doStartTag()**
  - The tag handler invokes this method when a request is made to a tag or when the starting tag of the element is encountered.
  - It returns either of the two field constants, returns the `SKIP_BODY` constant if there is no body to evaluate or returns the `EVAL_BODY_INCLUDE` constant.
  - **Syntax:**

```
public int doStartTag() throws JspException
```

- **setPageContext()**
  - This sets the current page context.
  - This is called by the page implementation prior to `doStartTag()`.
  - **Syntax:**

```
public void setPageContext(PageContext pc)
```

Methods of Tag Interface 2-3

- **doEndTag()**
  - The tag handler invokes this method when the JSP page encounters the end tag.
  - It generally follows the execution of the `doStartTag()` if the tag is empty.
  - This method returns the `SKIP_PAGE` constant if there is no need to evaluate the rest of the page.
  - It returns `EVAL_PAGE` constant if the rest of the page needs to be evaluated.
  - **Syntax:**

```
public int doEndTag() throws JspException
```

- **release()**
  - The container calls this method on the handler class when the tag handler object is no longer required.
  - **Syntax:**

```
public void release()
```

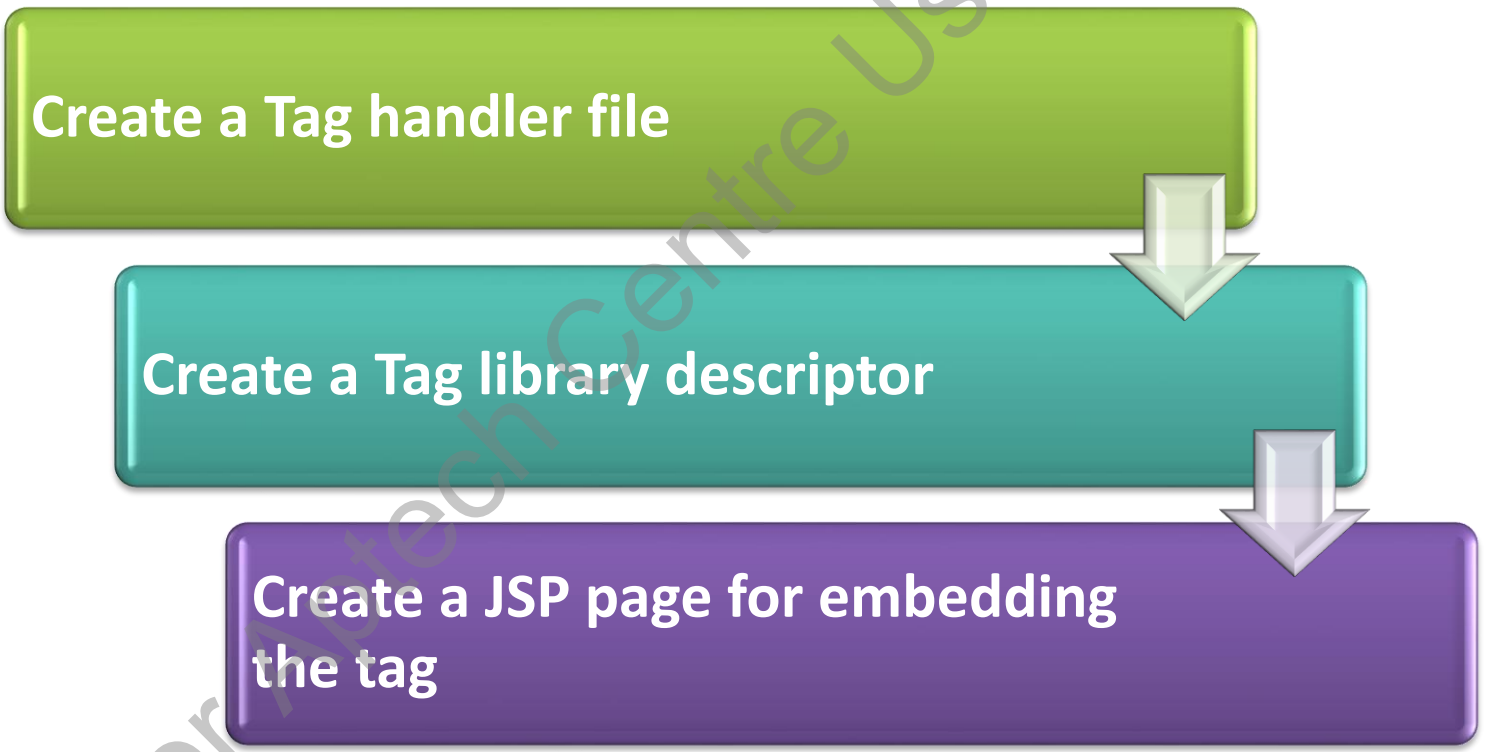
Methods of Tag Interface 3-3

- The code snippet demonstrates the methods of the `Tag` interface.

```
//The evaluation of start tag starts
public int doStartTag() throws JspException {
  try {
        pageContext.getOut().print("Creating Custom Tag by
        Implementing Tag Interface");
} catch (IOException ex) {
ex.printStackTrace();
}
        // The SKIP_BODY is returned
            return SKIP_BODY;
        }
}
//The evaluation of end tag starts
public int doEndTag() throws JspException {
        // the SKIP_PAGE is returned
            return SKIP_PAGE;
    }
//All the resources held by the tag handler is released
    public void release() {
    }
}
//The current value of the pageContext is set
public void setPageContext(PageContext pc) {
    this.pageContext = pc;
  }
//All the resources held by the tag handler is released
    public void release() {
    }
```

Empty Tags 1-5

- Empty tags are the simple tags without body content.
- To create an empty classic custom tag by implementing the `Tag` interface, follow the three basic steps:



Empty Tags 2-5

- **Create a Tag handler file**
  - The tag handler file that defines an empty classic custom tag implements the `Tag` interface.
  - The code snippet demonstrates how to create a tag handler class for the empty tag.

```
public class ImplementTag implements Tag {

     private PageContext ;
     private Tag parent;

//The current value of the pageContext is set
public void setPageContext(PageContext pc) {
     this.pageContext = pc;
   }

//The instance of the parent tag is set here.
public void setParent(Tag tag) {
this.parent = tag;
}


//The instance of the parent tag is retrieved here.
public Tag getParent() {
      return parent;
     }
```

Empty Tags 3-5

```
//The evaluation of start tag starts
public int doStartTag() throws JspException {
        try {
pageContext.getOut().print("Creating Custom Tag by
implementing Tag interface");
        } catch (IOException ex) {
        ex.printStackTrace();
        }

// The SKIP_BODY is returned
    return SKIP_BODY;
     }

//The evaluation of end tag starts
public int doEndTag() throws JspException {
// the SKIP_PAGE is returned
    return SKIP_PAGE;
     }

 //All the resources held by the tag handler is released.
    public void release() {
    }
}
```

Empty Tags 4-5

- **Create a Tag library descriptor**
  - We will create entry in tag library descriptor file for the empty.
  - The code snippet shows TLD file that will provide a detailed description of the Tag library and the custom tag.

```
<tag>
<name>Implement</name>
<tag-class>mytag.ImplementTag</tag-class>
 <body-content>empty</body-content>
</tag>
```

Empty Tags 5-5

- **Create a JSP page for embedding the tag**
  - In the final step, the code snippet shows that tag needs to be embedded in a JSP file.

```
<%--The tag is imported by the following directive.--%>
<%@taglib uri="/WEB-INF/tlds/TagInterface.tld" prefix="taginterface"%><HTML>
<HEAD></HEAD>
<BODY>
<%--The custom tag is called here.--%>
<taginterface:ImplementTag></taginterface:Implement>
</BODY>
</HTML>
```

Empty Tag to Accept Attribute 1-4

- The steps to create a classic custom tag to accept attribute are as follows:
  - **Create a tag handler file**
    - The tag handler class for the classic custom tag that accepts attribute need to implement the `Tag` interface.
    - Attributes are passed from the tag handler class to the tag inside the JSP page.
    - Only the start and end tags of the custom tag needs to be processed.
    - The code snippet demonstrates how to define attributes for the empty tag.

```
//The Tag interface is implemented to the TagAttribute class.
public class TagAttribute implements Tag {

private String name;

//The attribute, name is set by the setter method, setName()
public void setName(String name) {
this.name = name;
}

// The pageContext instance is set by the setPageContext() method
public void setPageContext(PageContext pc) {
    this.pageContext = pc;
   }
//The instance of the parent tag is retrieved here.
public Tag getParent() {
    return parent;
   }
```

Empty Tag to Accept Attribute 2-4

```
//The evaluation of start tag starts
public int doStartTag() throws JspException {
    try {
        pageContext.getOut().print(
        "This is my first tag with attribute!"+name);
    } catch (IOException ioe) {
        throw new JspException("Error:
        IOException while writing to client"
        + ioe.getMessage());
    }
    return SKIP_BODY;
    }
//The evaluation of end tag starts
    public int doEndTag() throws JspException {
        return SKIP_PAGE;
    }

//All resources held by the tag handler is released
    public void release() {
    }
}
```

Empty Tag to Accept Attribute 3-4

- **Create a Tag library descriptor**
  - We will create entry in tag library descriptor file for the empty.
  - The TLD file will provide a detailed description of the Tag library and the custom tag along with the attributes passed to it within the xml elements.
  - The code snippet shows the tag library descriptor.

```
<tag>
<name>welcomparam</name>
<tagclass>tags.TagAttribute</tagclass>
<bodycontent>empty</bodycontent>
<attribute>
<name>name</name>
<required>false</required>
<rtexprvalue>false</rtexprvalue>
</attribute>
</tag>
```

Empty Tag to Accept Attribute 4-4

- **Create a JSP page for embedding the tag**
  - In the final step, the code snippet tag needs to be embedded in a JSP file.

```
<%--The tag is imported by the following directive.--%>
<%@ taglib uri="/WEB-INF/jsp/TagAttribute.tld" prefix="first" %>
<HTML>
<HEAD>
</HEAD>
<BODY>
<%-- The tag is declared --%>
<first:welcomparam name="APTECH"/>
</BODY>
</HTML>
```

Non-empty Tags 1-4

- Tags that include body inside them are non-empty tags.
- We need to follow the three basic steps of creating a custom tag that has a body:
  - **Create a tag handler file:**
    - The tag handler class for the classic custom tag that has body content implements the `Tag` interface.
    - The `doStartTag()` method returns `EVAL_BODY_INCLUDE`.
    - The body is evaluated by the JSP container as any other java codes in the JSP page.

Non-empty Tags 2-4

- The code snippet demonstrates how to create the handler for the non-empty tag.

```
public class MyBodyTag implements Tag {
    private String name=null;
    private PageContext pc;
    private Tag pt;
    public void setName(String value){
        name = value;
    }
    public String getName(){
        return(name);
    }
    public void setPageContext(PageContext pageContext) {
        this.pc=pageContext;
    }
    public void setParent(Tag tag) {
        this.pt=tag;
    }
    public Tag getParent() {
        return pt;
```

Non-empty Tags 3-4

- **Create a Tag library descriptor**
  - The TLD file in this case will have the `<body-content>` as JSP, as shown in the code snippet.

```
<tag>
<name> MyBodyTag </name>
<tag-class>mytag.MyBodyTag </tag-class>
<body-content>JSP</body-content>
</tag>
```

Non-empty Tags 4-4

- **Create a JSP page for embedding the tag**
  - In the final step, the tag needs to be embedded in a JSP file, as shown in the code snippet.

```
<%--The tag is imported by the following directive.--%>
<%@taglib                uri="/WEB-INF/tlds/UsingTag.tld"
prefix="usingtag"%>
<HTML>
<HEAD>
<TITLE>Tag with Body</TITLE>
</HEAD>
<BODY>
    <%-- The tag is declared --%>
    <usingtag:Interface>
    <br>Current time: <%= new java.util.Date() %>
        </usingtag:MyBodyTag>
</BODY>
</HTML>
```

Methods of IterationTag Interface

- This interface has the method `doAfterBody()`.
- `doAfterBody()`:
  - Allows the user to conditionally re-evaluate the body of the tag.
  - Returns the constant `SKIP_BODY` or `EVAL_BODY_AGAIN`, if the `doStartTag()` method returns `EVAL_BODY_INCDLUDE`, then this method returns `EVAL_BODY_AGAIN` and the `doStartTag()` is evaluated once again.
  - When the `doAfterBody()` returns `SKIP_PAGE`, the `doEndTag()` method is invoked.
  - **Syntax:**

```
public int doAfterBody()throws JspException
```

Iterative Tag Sample

- The code snippet shows the `IterationTag` in a JSP page.

```
<%@ taglib prefix="iteration" uri="/WEB
INF/sampleIterationTagLib.tld" %>
<html><body>
  <iteration:loop count="5" >
       Hello IterationTag!</br>
  </iteration:loop>
</body></html>
```

Methods of BodyTag Interface

- The methods inside the `BodyTag` interface are as follows:
  - `setBodyContent()`
    - Sets the `bodyContent` object for the tag handler.
    - Encapsulates the body content of the custom tag for processing.
    - Is automatically invoked by the JSP page before the `doInitBody()` method.
    - **Syntax:**

```
public void setBodyContent(BodyContent b){ }
```

  - `doInitBody()`
    - Is invoked immediately after the `setBodyContent()` method returns the `bodyContent` object and before the first body evaluation.
    - **Syntax:**

```
public int doAfterBody()throws JspException
```

Sample BodyTag

- The code snippet shows the `BodyTag` in a JSP page.

```
<<%@ taglib prefix=”bodyTagPrefix” uri=”/WEB-
INF/sampleBodyTagLib. tld” %>
<html>
<body>
   <bodyTagPrefix:convertTempconvert=”C” >
   Temperature is:<%=temp%></br>
   </ bodyTagPrefix:loop>
</body>
</html>
```

Extending TagSupport and BodyTagSupport

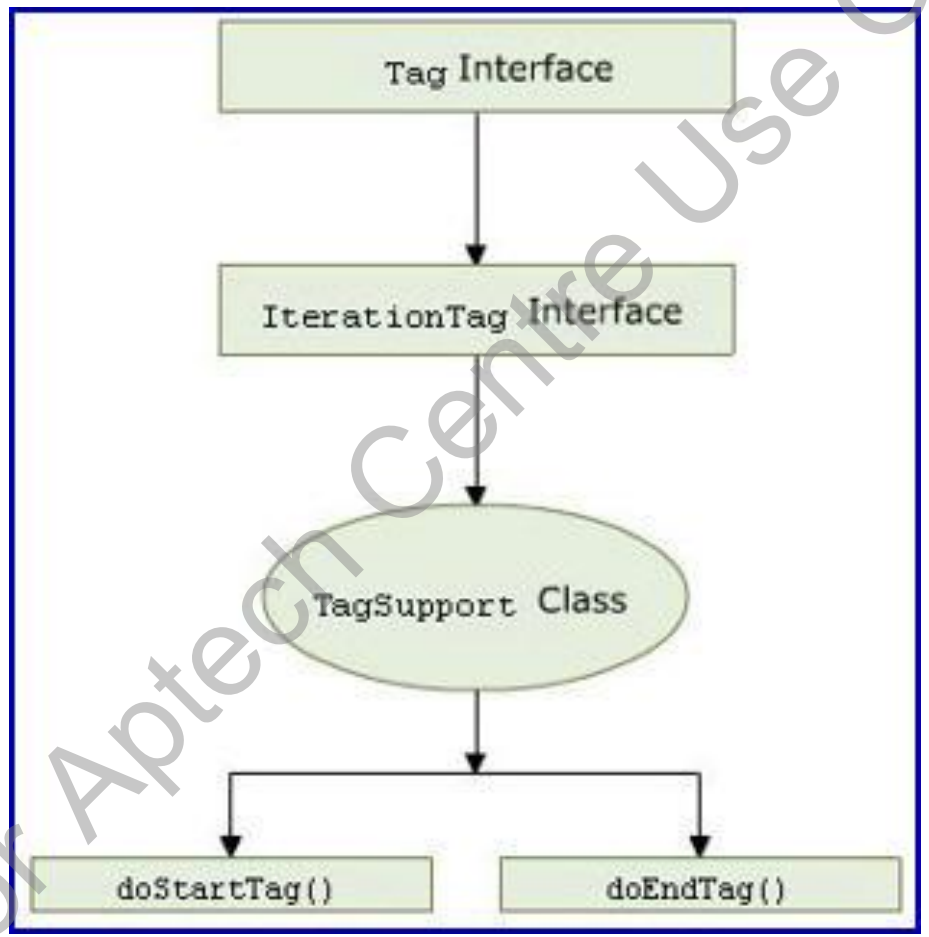
- The tag handler class needs to implement all the method declarations of the `Tag` or `IterationTag` interface when implementing these interfaces.
- In some case, we do not required to implement all these methods.
- The `TagSupport` class, which implements the `Tag` or `IterationTag` interface and provides a default implementation of all the methods.
- The programmer can extend `TagSupport` class and override the required method.

TagSupport Class 1-2

- The `TagSupport` class implements the `IterationTag` interface.
- It provides default implementations for each of the methods of the `Tag` and `IterationTag` interfaces.
- Some methods of `TagSupport` class are as follows:
  - `doStartTag()`: This method is inherited from `Tag` interface and by default, return value is `SKIP_BODY`.
  - `doEndTag()`: This method is inherited from `IterationTag` interface and by default, return value is `SKIP_BODY`.

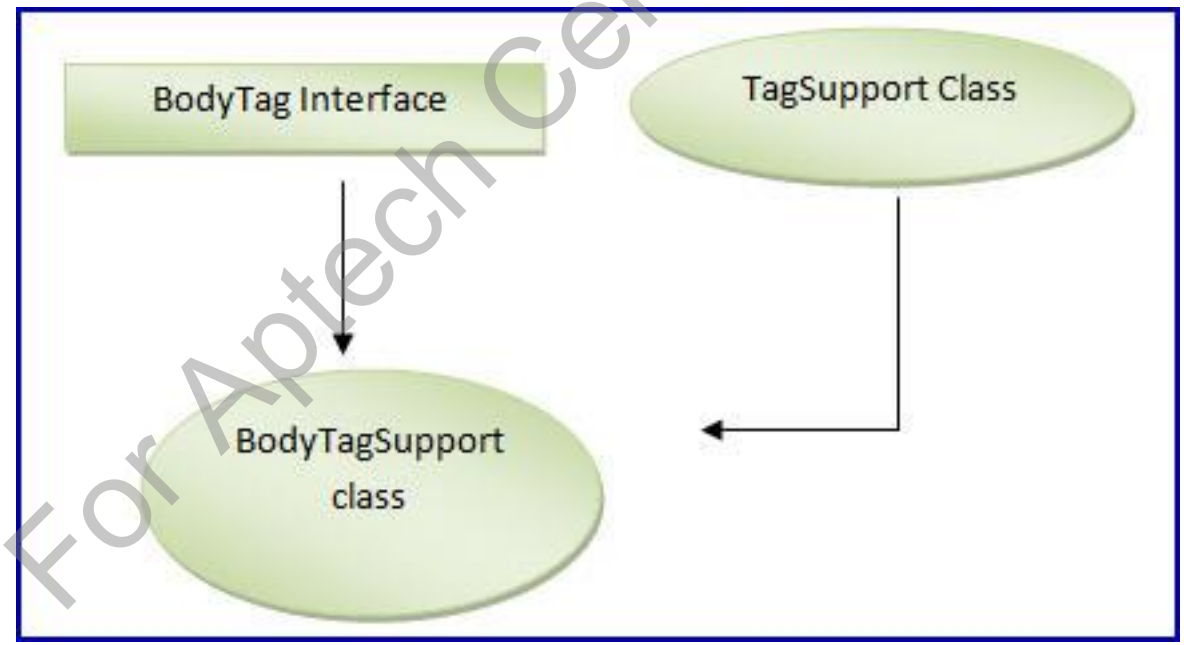
TagSupport Class 2-2

- Figure depicts the hierarchy of `TagSupport` class.



BodyTagSupport Class 1-2

- This class by default implements the `BodyTag` interfaces and also extends to the `TagSupport` class.
- The handler class only needs to override the methods of the `BodyTagSupport` class.
- All the methods of the `TagSupport` class and `BodyTag` interface are implemented by default in the handler class.
- Figure depicts the hierarchy of `BodyTagSupport` class.



BodyTagSupport Class 2-2

- The `BodyTagSupport` class contains the following overridden methods:
  - **`getBodyContent()`:**
    - Retrieves the `bodyContent` object of the class `BodyContent`.
    - Contains the body of the tag which is accessed by the container for manipulation.
    - **Syntax:**

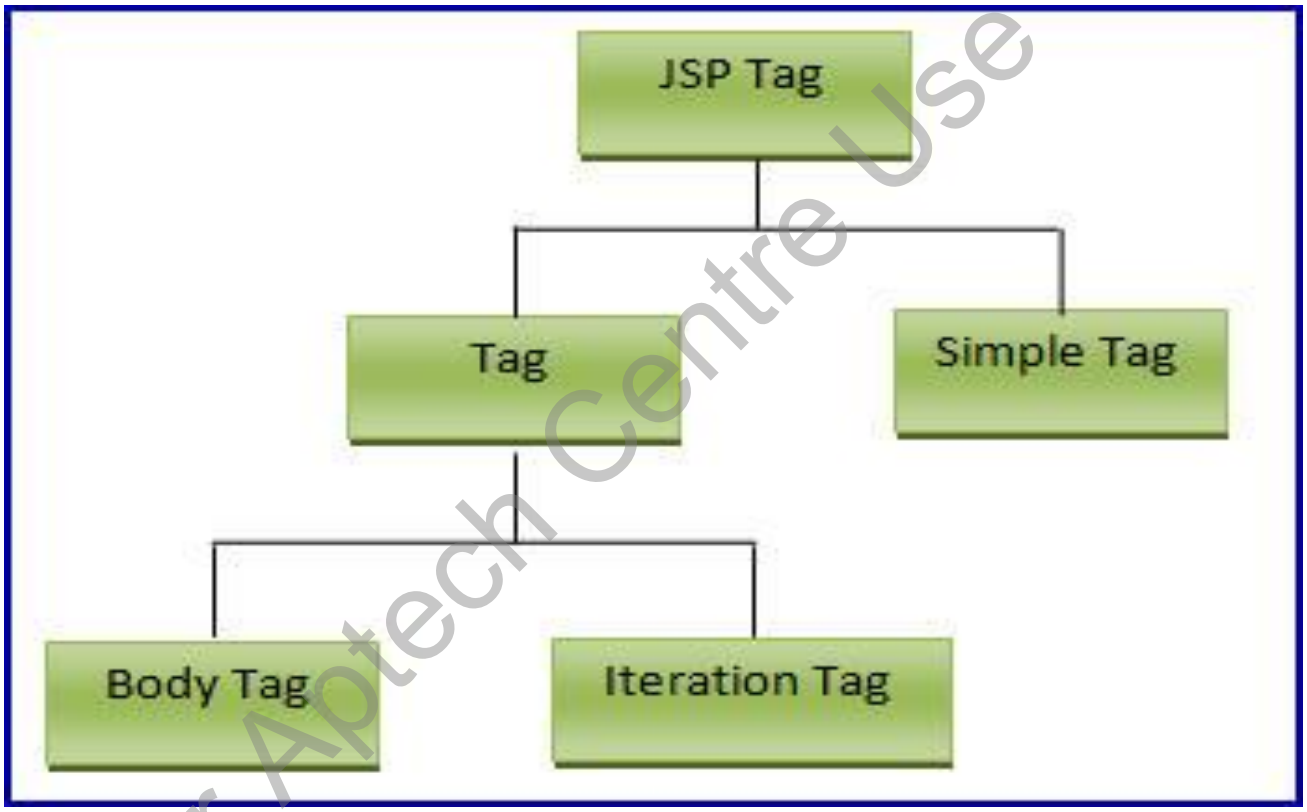
```
public BodyContent getBodyContent()
```

  - **`setBodyContent()`:**
    - Is set by the `setBodyContent()` method.
    - **Syntax:**

```
public void setBodyContent(BodyContent b)
```

SimpleTag 1-3

- Figure depicts the relationship between `SimpleTag` and other JSP tag interfaces.



SimpleTag 2-3

- Following methods are present in `SimpleTag` interface:
  - **`doTag()`**
    - Is called by the container to begin `SimpleTag` operation and used for handling tag processing.
    - Retains body iteration after being processed.
    - **Syntax:**

```
public void doTag() throws JspException,IOException
```

  - **`setParent()`**
    - Sets the parent of a specified tag.
    - **Syntax:**

```
public void setParent(JspTag parent)
```

  - **`getParent()`**
    - Returns the parent of the specified tag.
    - **Syntax:**

```
public JspTag getParent()
```

SimpleTag 3-3

- **`setJspContext()`**
  - Sets the context to the tag handler for invocation by the container.
  - **Syntax:**

```
public void setJspContext(JspContext pc)
```

- **`setJspBody()`**
  - Is provided by the body of the specified tag, makes the body content available for tag processing.
  - **Syntax:**

```
public void setJspBody(JspFragment jspBody)
```

SimpleTagSupport

- The `SimpleTagSupport` class acts as a base class for simple tag handlers.
- The class implements the **`SimpleTag`** interface.
- Some of the useful methods are as follows:
  - **`getJspContext()`**
    - Returns the context passed into the container by `setJspContext()` method.
    - **Syntax:**

```
protected JspContext getJspContext()
```

  - **`getJspBody()`**
    - Returns the `JspFragment` object for body processing in the tag.
    - **Syntax:**

```
protected JspFragment getJspBody()
```

Implementation of SimpleTag Interface 1-3

- The code snippet shows the interface that is implemented by extending the `SimpleTagSupport` class and overriding the `doTag()` method.

```
import javax.servlet.jsp.tagext.*;
import javax.servlet.jsp.*;
public class Greeter extends SimpleTagSupport {
    public void doTag() throws JspException {
        PageContext pageContext = (PageContext)
        getJspContext();
        JspWriter out = pageContext.getOut();
    }
}
```

Implementation of SimpleTag Interface 2-3

- The code snippet shows the corresponding tag in TLD file.

```
<tag>
<name>greeter</name>
<tag-class>Greeter</tag-class>
<body-content>empty</body-content>
<description>
    Print this on the browser.
</description>
</tag>
```

Implementation of SimpleTag Interface 3-3

- The code snippet shows the corresponding JSP file.

```
<%@ taglib prefix="ui" uri="/WEB-INF/sampleTag.tld "
%>
<html><body>
<ui: greeter />
</body></html>
```

Summary

- Custom tags are action elements on JSP pages that are mapped to tag handler classes in a tag library.
- Tag libraries allow us to use independent Java classes to manage the presentation logic of the JSP pages, thereby reducing the use of scriptlets and leveraging existing code to accelerate development time.
- The tag without any body is called an empty tag and the tag accepting attributes is called a tag with attributes or a parameterized tag.
- A tag with JSP code or tag dependent strings is called a tag with body. If a tag contains several other tags inside it then it is called a nested tag.
- The Tag Library Descriptor (TLD) file contains the information that the JSP engine needs to know about the tag library in order to successfully interpret the custom tags on JSP pages.
- The SimpleTag interface provides the doTag() method, which is supported by the Classic Tag Handlers.