The "Classic" Custom Tag Event Model

1. Are there any classes in the javax.servlet.jsp.tagext package that already implements Tag, IterationTag or BodyTag?
    a. There isn't a class that directly implements the **Tag** interface.
    b. **TagSupport** implements **IterationTag** which in turn extends from Tag interface.
    c. **BodyTagSupport** implements **BodyTag** in turn extends from IterationTag interface.
2. List down the signatures of methods that need to be implemented while implementing the **Tag interface**?
    a. public void **setPageContext**(PageContext pageContext) - the container set the page context.
    b. **public void setParent(Tag parent)** – Container invokes this method to set the Parent Tag of this Tag in this document.
    c. public Tag **getParent**()
    d. If there are attributes, there should be corresponding setter methods following javabean conventions. Signature for an attribute beforeBody:
        i. public void setBeforeBody(boolean beforeBody)
    e. public int **doStartTag**() throws JspException
    f. public int **doEndTag**() throws JspException
    g. public void **release**()
3. Explain the use of doStartTag() and doEndTag() methods in the classic custom tag event life cycle?
    a. The **doStartTag**() is called when the JSP container processes the appearance of Tag implementing class's opening tag within the JSP page source. The doStartTag() return an int value to tell the JSP container what to do next. Valid values are defined as public final static int data members either within the Tag interface or the BodyTag interface:
        i. Tag.**EVAL_BODY_INCLUDE**—body for this action should now be processed.
        ii. Tag.**SKIP_BODY**—body for this action should be ignored and proceed directly to the doEndTag() method for this action.

iii. BodyTag.**EVAL_BODY_BUFFERED** - make the **BodyContent** object available. The container will now invoke setBodyContent(BodyContent) and doInitBody() if the body is not empty. Only allowed if a BodyTag interface is implemented.

b. The **doEndTag**() is called when the JSP container processes the appearance of the ending tag within the JSP page source. The two possible return types are:

i. Tag.**EVAL_PAGE** tells the JSP container to process the rest of the page after the closing tag.

ii. Tag.**SKIP_PAGE** effectively tells the JSP container to abort the rest of the page following the closing tag.

4. Explain the use of the **release**() method in the classic custom tag event life cycle?
   a. If the container decides to take a particular instance of a tag out of service, then release() is called, which you can use to clean up any expensive resources associated with the tag.
5. Will the **release**() method be called every time a custom tag appears in a JSP page?
   a. No, this won't happen every time the tag appears in a JSP page. The instance of the tag is potentially reused over and over again.
6. During the classic custom tag life cycle, why you can't rely on the constructor to do initialization that must take place for each specific use?
   a. Instances of tag handler classes are reused. So you don't necessarily get a new instance with every use of a particular tag on your JSP page, or even across JSP pages.
7. Give the Tag life cycle methods which are executed for each appearance of the tag on the page?
   a. Apart from the no-argument constructor and release(), all other Tag life cycle methods are executed for each appearance of the tag on the page. One good way is to use setPageContext() for that initialization that must take place for each specific use.
8. What is the **classic custom tag event life cycle** for a Tag interface?
   a. The JSP container makes an **instance** of the Tag implementing class.
   b. It then calls **setPageContext**(PageContext pc).

c. Then it calls **setParent**(Tag parent).
d. Now the JSP container calls any other set methods on Tag implementing class that relates to **attributes** for the tag.
e. Next the JSP container calls **doStartTag**() and then **doEndTag**().
f. Finally, the JSP container calls **release**().

9. After writing a classic custom tag handler class, what all needs to be done before we can use the tag within a page?
    a. We must define the tag within a TLD file associating it with the class.
    b. Configure the TLD file in web.xml.
    c. Set up a taglib directive in the JSP page and then use it.
10. How do we define a tag within a TLD file for a custom tag handler class?
    a. Each tag must have a **<tag>** element defined. The tag must have at least the three sub-elements as:
        i. **<name>:** a unique name within the tag library.
        ii. **<tag-class>:** the fully qualified name of the tag handler class.
        iii. **<body-content>:** the sort of content permitted in the body of the tag - empty, scriptless, tagdependent or JSP.
    b. If there are attributes, the tag element will have an **<attribute>** sub element which contains three mandatory subelements:
        i. **<name>—**a unique name for the attribute within each tag.
        ii. **<required>—**if set to "true," the attribute must appear in the opening tag. The default is false.
        iii. **<rtexprvalue>—**if set to "true" a run-time (EL) expression value is permitted for the attribute's value setting. The default is false.
    c. Example:
        i. <taglib ...taglib attributes...>
        ii. <... elements omitted ...>
        iii. **<tag>**
        iv. **<name>**dateStamp1</name>
        v. **<tag-class>**webcert.ch08.examp0804.DateStampTag1</tag-class>
        vi. **<body-content>**empty</body-content>

vii. **<attribute>**
viii. **<name>**beforeBody</name>
ix. **<required>**false</required>
x. **<rtexprvalue>**true</rtexprvalue>
xi. </attribute>
xii. </tag>
xiii. </taglib>

11. A TLD entry has <name>dateABC</name> and <tag-class>web.abc.DateTag1</tag-class>. In the JSP it is associated with a prefix mytags. Give an usage in the page?
    i. <mytags: dateABC />
12. Can you set an attribute value for a custom tag using an EL expression?
    a. Yes. However <rtexprvalue> sub element of <attribute> must be true.
        i. <c:set var="trueVariable" value="true" />
        ii. <p><mytags:dateStamp2 beforeBody="${trueVariable}"/></P>
13. What is the classic custom tag event life cycle for an IteratorTag interface?
    a. The JSP container makes an **instance** of the Tag implementing class.
    b. It then calls **setPageContext**(PageContext pc).
    c. Then it calls **setParent**(Tag parent).
    d. Now the JSP container calls any other set methods on Tag implementing class that relates to **attributes** for the tag.
    e. Next the JSP container calls doStartTag(), **doAfterBodyTag**() and then doEndTag().
    f. Finally, the JSP container calls release().
14. The doAfterBody() method belongs to which interface – Tag or IteratorTag?
    a. IteratorTag.
15. Describe the purpose and usage of doAfterBody() method of IteratorTag interface?
    a. Following doStartTag() call, the container calls doAfterBody().
    b. The JSP container calls this method after evaluating the body but before doEndTag(). The method returns two possible values:
        i. Tag.**SKIP_BODY** - The JSP container won't process any further occurrences of the body, but proceed directly to the doEndTag().

ii. IterationTag.**EVAL_BODY_AGAIN** - The JSP container will evaluate the body of the tag again, and then once again invoke the doAfterBody().

16. The doStartTag() method ends by returning Tag.EVAL_BODY_INCLUDE and the doAfterBody() method returns Tag.SKIP_BODY. Will the body be executed?
    a. Tag.SKIP_BODY from doAfterBody() means that the JSP container won't process any further occurrences of the body, but proceed directly to the doEndTag() method. However the doStartTag() returning Tag.EVAL_BODY_INCLUDE ensures that the body is processed at least once, for the doAfterBody() occur for the first time only after the body has been processed for the first time.
17. While using a custom tag in our JSP, should we handle multi-threading issues?
    a. No. The JSP container must guarantee that any given instance of a tag handler class is dedicated to one thread at one given moment.
18. Why is it better to extend TagSupport than implement the IteratorTag directly?
    a. TagSupport implements the IterationTag interface and provides a default implementation of all the Tag and IterationTag methods, and add a few useful methods of its own.
    b. So effort for writing a tag class that extends TagSupport will be less than implementing IterationTag from scratch. Here you have to override only those methods where the default TagSupport behavior is insufficient.
19. Describe the use and usage of **findAncestorByClass**() method of the TagSupport class?
    a. The findAncestorByClass() accepts two parameters: the first an instance of the tag whose ancestor is sought, and the second the class type of the ancestor tag. An example code snippet in the doEndTag() method:
        i. Class parentClass;
        ii. try {
        iii. parentClass = Class.forName("examp0804.AnyParentTag");
        iv. } catch (ClassNotFoundException e1) {
        v. throw new JspException(e1);
        vi. }
        vii. AnyParentTag pTag = (AnyParentTag)

viii. TagSupport.findAncestorWithClass(this, parentClass);

b. Here findAncestorWithClass will return the expected parent tag or a subclass of it, even if it is not the immediate parent and avoid a ClassCastException. The first (closest) relative is returned.

c. Under the covers, the code makes use of the getParent() method for each tag examined in the hierarchy.

20. Give any validation mechanism while using custom Tags?

a. You can write a class extending javax.servlet.jsp.tagext.**TagLibraryValidator**. The **validate**() method of **TagLibraryValidator** which will be called by the container at translation time, passes in an XML representation of the entire JSP page to be validated, giving you the opportunity to perform cross-tag checking and—if necessary—halting the translation process with an appropriate error message.

21. Describe the **setId**(String id) and **getId**() methods of the TagSupport Class?

a. public void setId(String id) – Set the id attribute for the Tag.

b. public String getId() – Get the value of the id attribute of this tag; or null.

c. These methods assume that you will set up an attribute called id for your tag, intended to uniquely identify the tag on the page and only if you do, then the JSP container will invoke the setId() method in the normal way and we will get the id by invoking getId().

22. How is the handling of the setId(String id) and getId() methods different from normal attributes?

a. Need to find the answer

23. Describe the following methods of the TagSupport class - public void **setValue**(String name, Object o), public Object **getValue**(String name), public void **removeValue**(String name), and public Enumeration **getValues**()?

a. These methods use an instance variable Hashtable within the TagSupport class:

i. **setValue()** associate named values with the tag instance

ii. **getValue**() get them back according to their name.

iii. **getValues**() method returns an Enumeration of keys. You can use each key in turn with the getValue() method to return the underlying object.

iv. **removeValue() –** removes values.

24. Since we have setValue(String name, Object o) method in TagSupport class, do we still have to write setters for attributes?
    a. The setValue() method in TagSupport is not a substitute for "setter" methods for attribute. The JSP container will not call the setValue() method; it's there for your own internal use in the tag handler code.
25. Give an advantage of the use of the IterationTag over the use of JSTL's iteration capabilities?
    a. We can use the IterationTag life cycle when we have complex business functionality to incorporate. Encapsulation is usually more readily achieved inside tag handler code than when using naked JSTL within JSP page source.
26. What does a BodyTag adds to an IteratorTag?
    a. The additional capability that a BodyTag has over its predecessors is the ability to manipulate the content of its body.
27. List down the new life cycle methods of the BodyTag over and above those provided by Tag and IterationTag?
    a. **setBodyContent**(BodyContent)
        i. Gives you the opportunity to save the BodyContent object.
        ii. Called after doStartTag(), but before the next method, doInitBody().
    b. **doInitBody**()
        i. Called before the JSP container enters and evaluates this tag's body for the first time. It is called only once per access to the tag and not every time you iterate through the body. At this point, the BodyContent object—though available—is empty: The body hasn't been evaluated at all yet.
28. You need to do some pre-tag operations on the BodyContent object like writing some initial content to the stream before evaluation of action's BodyContent takes place. Which method of the BodyTag interface you will use for this?
    a. **doInitBody**(). The doInitBody() method is called before the JSP container enters and evaluates this tag's body for the first time. It is called only once per each access to the tag. At this point, the BodyContent object—though available—is empty: The body hasn't been evaluated at all yet.

29. Why can't we use doStartTag() method to do some pre-tag operations on the BodyContent while using the BodyTag interface?
    a. **setBodyContent**() is called after doStartTag() which save the BodyContent object that will hold the buffered body contents. Therefore doStartTag() is invoked before the BodyContent buffer has been established and cannot be used to do some pre-tag operations on the BodyContent.
30. What is the classic custom tag event life cycle for a **BodyTag** interface?
    a. The JSP container makes an **instance** of the BodyTag implementing class.
    b. It then calls **setPageContext**(PageContext pc).
    c. Then it calls **setParent**(Tag parent).
    d. Now the JSP container calls any other set methods on Tag implementing class that relates to **attributes** for the tag.
    e. Next the JSP container calls doStartTag(), **setBodyContent**() , **doInitBody**(), doAfterBodyTag() and then doEndTag().The doInitBody() method is called only once per each access to the tag.
    f. Finally, the JSP container calls **release**().
31. What does it signify if the doStartTag() returns BodyTag.**EVAL_BODY_BUFFERED**?
    a. Returning BodyTag.EVAL_BODY_BUFFERED results in container calling the setBodyContent() and doInitBOdy() methods and make the BodyContent object available.
32. Suppose you have a JSP page with such a tag setup: <mytags:sqlExecute>SELECT * FROM PRODUCT WHERE NAME LIKE %1</mytags:sqlExecute>. The user never wants to see the query, but a formatted set of rows in an HTML table, showing details about a selection of products from the appropriate database table. Sketch out how you might approach writing a BodyTag to perform this translation.
    a. Ensure that you pass back EVAL_BODY_BUFFERED from the doStartTag() method (in BodyTagSupport this is the default return code).
    b. Trap the BodyContent object in a private instance variable in your implementation of the setBodyContent() method (done already in BodyTagSupport).

c. Override doAfterBody(), calling the getString() method on the Body Content object—return this to a local String variable. This variable will contain the SQL string as an outcome of evaluating the body.
d. Still in doAfterBody(), establish a data source connection to your database, and execute the SQL string. From the ResultSet returned, format the output as desired (in an HTML table, XML etc).
e. Write this to page output using the JspWriter returned by the BodyContent object's getEnclosingWriter() method.
f. Still in doAfterBody(), release your data source connection, and return SKIP_BODY to prevent any further evaluations.

33. What is a BodyContent Object?
    a. The BodyContent object effectively traps the output from any evaluations of the body that occur within the tag.
    b. The BodyContent is a JspWriter, so you can write additional content to the BodyContent object in your tag handler code, before, between, and after evaluations of the tag body that arrive in the BodyContent object as well.
    c. Although the BodyContent object is a writer, there is no real underpinning stream. Usually when you are finished doing all you need to evaluate in the tag, most likely at the end of doEndTag(), we can redirect the output to some other writer. BodyContent's **writeOut**(Writer w) method can redirect the output to a writer w.
34. Although the BodyContent object is a writer, there is no real underpinning stream. What does it mean?
    a. There is no real underpinning stream for BodyContent; everything that accumulates there is held in a string buffer. Whatever you write to BodyContent won't arrive in the page output until you redirect it to some other writer. BodyContent's **writeOut**(Writer w) method can redirect the output to a writer w.
35. What is the effect of calling clearBody() on the body content object?
    a. Clears the BodyContent buffer, causing all its contents to be lost.
36. What is the effect of calling flush() on the body content object?
    a. Throws an IOException. We can't flush the buffer since it is not backed by another stream.

37. What is the use of calling **getEnclosingWriter**() on the body content object?
    a. **BodyContent's getEnclosingWriter**() passes back a JspWriter object, mostly the JspWriter associated with the JSP page. If your tag is nested within another BodyTag, the enclosing writer could be another BodyContent object.
38. What is the use of calling **getReader**() on the body content object?
    a. Returns a Reader suitable for extracting the contents of this buffer.
39. What is the use of calling **getString**() on the body content object?
    a. Returns the contents of the BodyContent buffer as a string.
40. Can we use the out implicit object as an argument to **BodyContent's writeOut**(Writer w) method?
    a. Yes. The out implicit object has a type of jsp.JspWriter. However this might bypass the parent writer and may lead to incorrect behavior such as not evaluating a conditional parent tag.

MORE NOTES

1. All custom tags – classic and simple – should implement **javax.servlet.jsp.tagext.JspTag** interface. Both Tag (for classic custom) and SimpleTag (for simple tags) interfaces extend from JspTag.
2. Most of the tag related classes and interfaces reside in the **javax.servlet.jsp.tagext** package.
3. In the classic custom tag event life cycle there are **three possibilities** you can choose from, based on three interfaces. **Tag**, **IterationTag**, and **BodyTag**. All three are in the **javax.servlet.jsp.tagext** package.
4. Tag, IterationTag, and BodyTag extend each other.
    a. **Tag** is the parent interface.
    b. **IterationTag** extends **Tag**, adding some looping capabilities.
    c. **BodyTag** extends **IterationTag**, adding further method calls to manipulate the body content of the tag.
5. The JSP container makes an instance of the Tag implementing class. The class must have a no-argument constructor—either explicitly defined or provided by the compiler in the absence of other constructors.
6. The classic custom tag life cycle method **setPageContext**(PageContext pageContext) gives the tag an opportunity to save a reference to the page context of the JSP page that contains the tag. The page context can be used to get all the other implicit objects available to a JSP page.
7. If the doStartTag() does not return BodyTag.EVAL_BODY_BUFFERED the setBodyContent() and doInitBody() will not be called.
8. The right thing to do with the BodyContent object is to pass the output to the enclosing writer, for whatever logic is controlling the enclosing writer should have a chance to influence what happens to the output produced in your inner tag.
9. While using IterationTag interface, if the tag body is empty, irrespective of the return type of doStartTag() execution jumps straight from doStartTag() to doEndTag() skipping doAfterTag(). Need to try.

10. The default return values from the methods of TagSupport class skip the body, prevent further evaluations and continue with evaluation of the calling JSP:
    a. doStartTag() returns Tag.**SKIP_BODY**
    b. doAfterBody() returns Tag.**SKIP_BODY**
    c. doEndTag() returns Tag.**EVAL_PAGE**
11. The default return values from the methods of BodyTagSupport class evaluate the body only once, using the BodyContent buffer and continue with evaluation of the calling JSP:
    a. doStartTag() returns BodyTag.**EVAL_BODY_BUFFERED**
    b. doAfterBody() returns Tag.**SKIP_BODY**
    c. doEndTag() returns Tag.**EVAL_PAGE**
12. While using BodyTag interface, setBodyContent(BodyContent) and doInitBody() are invoked only if doStartTag() return BodyTag.**EVAL_BODY_BUFFERED** and the body content of the action in the JSP is not empty.
13. The BodyContent class is a sub class of JspWriter which acts as a buffer not backed by any other stream. As a result, discarding its contents causes them to be omitted from the output stream.
14. The basic things the JSP container do while it processes an occurrence of a tag implementing the Tag / IterationTag / BodyTag interfaces in a JSP page is called **classic custom tag event life cycle** for a Tag / IterationTag / BodyTag interface.
15. BodyContent object has a method for redirecting the output to some writer: **writeOut**(Writer w). The entire contents of the BodyContent object are appended to the Writer of your choice.
16. A typical invocation you see at the end of your tag logic that use BodyContent's **getEnclosingWriter**() method goes like this:
    a. BodyContent bc = getBodyContent();
    b. bc.writeOut(bc.getEnclosingWriter());
17. A tag library can register Servlet Context event listeners in case it needs to react to such events. Listener entries look like this:
    a. <listener>
    b. <listener-class>com.mycompany.TagLibListener</listener-class>
    c. </listener>