Chapter 10

How to use custom JSP tags

Objectives

Applied

1. Create a Tag Library Descriptor (TLD) for custom tags, and write the tag handler classes that implement these tags.
2. Use custom JSP tags in the JSPs for your applications.

Knowledge

1. Explain how a custom JSP tag gets associated with a tag handler class.

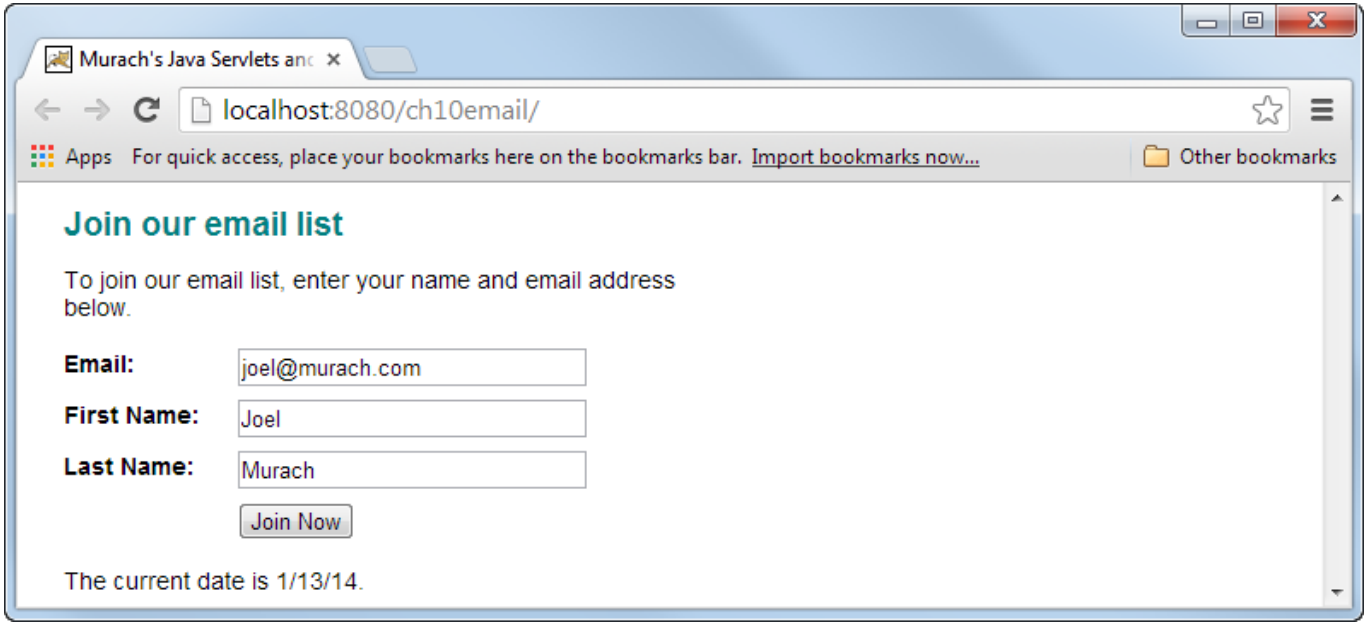
A taglib directive for a custom tag library

```
<%@ taglib prefix="mma" uri="/WEB-INF/murach.tld" %>
```

A JSP that uses a custom tag

```
<p>The current date is <mma:currentDate />.</p>
```

JSP that displays the custom tag



TLD file with two tag elements

```
<?xml version="1.0" encoding="UTF-8"?>
<taglib version="2.0" xmlns="http://java.sun.com/xml/ns/j2ee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
       web-jsptaglibrary_2_0.xsd">

   <tlib-version>1.0</tlib-version>
   <short-name>murach</short-name>
   <uri>/WEB-INF/murach.tld</uri>
   <info>A custom tag library developed by
         Mike Murach and Associates</info>

   <tag>
      <name>currentDate</name>
      <tagclass>murach.tags.CurrentDateTag</tagclass>
      <info>Returns the current date with the SHORT date format</info>
    </tag>

   <tag>
      <name>currentTime</name>
      <tagclass>murach.tags.CurrentTimeTag</tagclass>
   </tag>

</taglib>
```

Custom tags

- The *Tag Library Descriptor* (*TLD*) is an XML document that describes a *tag library* with custom tags.
- An application typically uses a single TLD to define all of its custom tags.
- There's no limit to the number of TLDs an application can have.
- Within a tag element, you must use the name element to specify the name of the custom tag.
- Within a tag element, you must use the tagclass element to specify the *tag class* for the tag.
- Within a tag element, you can optionally use the info element to specify descriptive information about the tag.
- The elements that are required by a TLD may vary depending on the JSP engine.

A custom tag that doesn't have a body

```
package murach.tags;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;
import java.util.*;
import java.text.DateFormat;

public class CurrentDateTag extends TagSupport {
    @Override
    public int doStartTag() throws JspException {
        Date currentDate = new Date();
        DateFormat dfs = DateFormat.getDateInstance(DateFormat.SHORT);
        String currentDateFormatted = dfs.format(currentDate);

        try {
            JspWriter out = pageContext.getOut();
            out.print(currentDateFormatted);
        } catch (IOException ioe) {
            System.out.println(ioe);
        }
        return SKIP_BODY;
    }
}
```

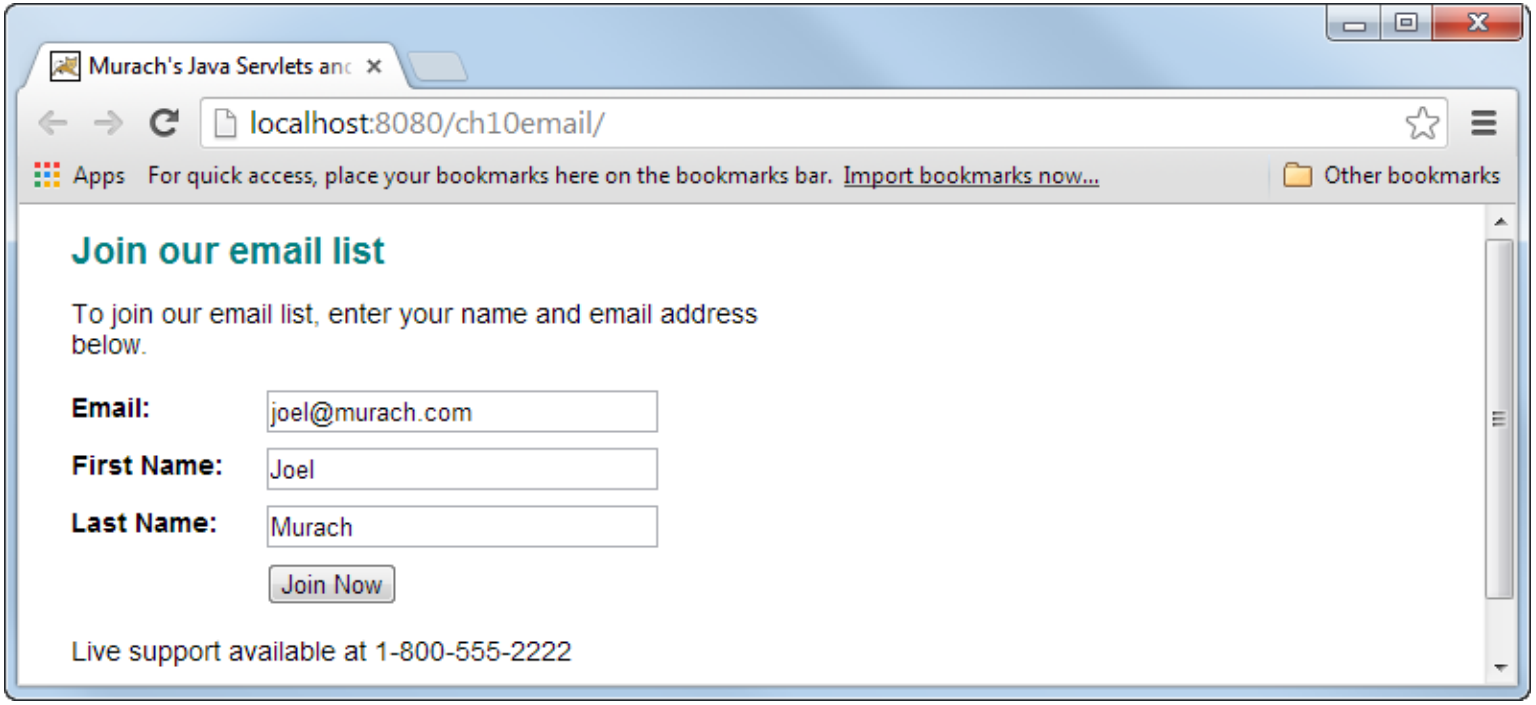
A custom tag that doesn’t have a body

- The *tag class*, or *tag handler class*, is the Java class that defines the actions of the tag.
- A tag class must implement the Tag interface.
- For a tag that doesn’t have a body, implement the Tag interface by extending the TagSupport class and overriding the doStartTag method.
- To display text on the JSP, use the print method of the JspWriter object.
- To get a JspWriter object, use the getOut method of the pageContext object that’s defined in the TagSupport class.
- For a tag that doesn’t have a body, the doStartTag method must return the SKIP_BODY constant.

A custom tag with a body

```
<mma:ifWeekday>
   <p>Live support available at 1-800-555-2222</p>
</mma:ifWeekday>
```

A JSP that displays the tag Monday through Friday



The tag element in the TLD file

```
<tag>
    <name>ifWeekday</name>
    <tagclass>murach.tags.IfWeekdayTag</tagclass>
    <bodycontent>JSP</bodycontent>
</tag>
```

The tag class

```
package murach.tags;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.util.*;

public class IfWeekdayTag extends TagSupport {

    @Override
    public int doStartTag() throws JspException {

        Calendar currentDate = new GregorianCalendar();
        int day = currentDate.get(Calendar.DAY_OF_WEEK);
        if (day == Calendar.SATURDAY || day == Calendar.SUNDAY) {
            return SKIP_BODY;
        } else {
            return EVAL_BODY_INCLUDE;
        }
    }
}
```

A custom tag with a body

- A tag that has a body must have an opening tag, a body, and a closing tag.
- The body of the tag can contain any HTML or JSP elements.
- The tag class for a custom tag can control whether the body of the tag is displayed in the JSP.
- When you add a tag that has a body to a TLD, you must specify a value of “JSP” for the bodycontent element.
- To create a tag class for a tag that has a body, you extend the TagSupport class and override the doStartTag method.
- To display the body of the tag in the JSP, the tag class should return the EVAL_BODY_INCLUDE constant. Otherwise, the tag class should return the SKIP_BODY constant.

How to use a custom tag that has attributes

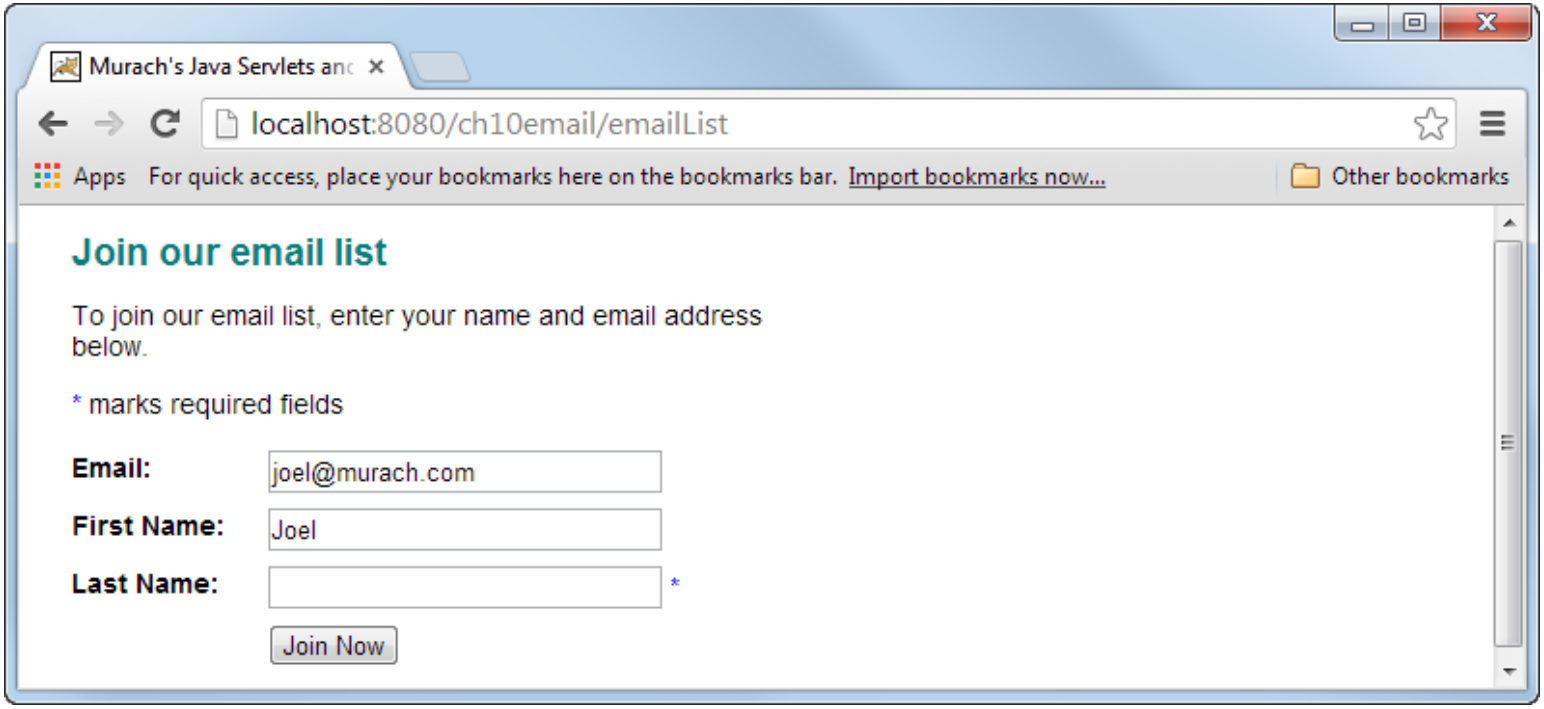
To display the asterisk

```
<p><mma:ifEmptyMark color="blue" field=""/> marks required fields</p>
```

To display the asterisk only if a field is empty

```
<label class="pad_top">Last Name:</label>
<input type="text" name="lastName" value="${user.lastName}">
<mma:ifEmptyMark color="blue" field="${user.lastName}"/><br>
```

A JSP that uses a custom tag that has attributes



Syntax for the attribute element in a tag element

```
<attribute>
    <name>attributeName</name>
    <required>true|false|yes|no</required>
    <rtexprvalue>true|false|yes|no</rtexprvalue>
    <type>data_type</type>
</attribute>
```

A tag element with two attributes

```
<tag>
    <name>ifEmptyMark</name>
    <tagclass>murach.tags.IfEmptyMarkTag</tagclass>
    <bodycontent>empty</bodycontent>
    <attribute>
        <name>color</name>
        <required>false</required>
    </attribute>
    <attribute>
        <name>field</name>
        <required>true</required>
        <rtexprvalue>true</rtexprvalue>
    </attribute>
</tag>
```

A tag class that uses two attributes

```
package murach.tags;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;

public class IfEmptyMarkTag extends TagSupport {

    private String field;
    private String color = "red";

    public void setField(String field) {
        this.field = field;
    }

    public void setColor(String color) {
        this.color = color;
    }
```

A tag class that uses two attributes (continued)

```
@Override
    public int doStartTag() throws JspException {
        try {
            JspWriter out = pageContext.getOut();
            if (field == null || field.length() == 0) {
                out.print("<font color=" + color + "> *</font>");
            }
        } catch (IOException ioe) {
            System.out.println(ioe);
        }
        return SKIP_BODY;
    }
}
```

The attribute child elements

Element	Description
`<name>`	Name of the attribute.
`<required>`	True/false value that specifies whether this attribute is required. If it isn't required, the tag class should provide a default value.
`<rtexprvalue>`	True/false value that specifies whether the value of the attribute is determined from a runtime expression. If so, the type element can be any data type. Otherwise, the type element is a string.
`<type>`	Data type of the attribute value. Code this element when the value of the attribute is determined from a runtime expression and the data type isn't a string.

An attribute element that uses the integer data type

```
<attribute>
    <name>count</name>
    <required>true</required>
    <rtexprvalue>true</rtexprvalue>
    <type>int</type>
</attribute>
```

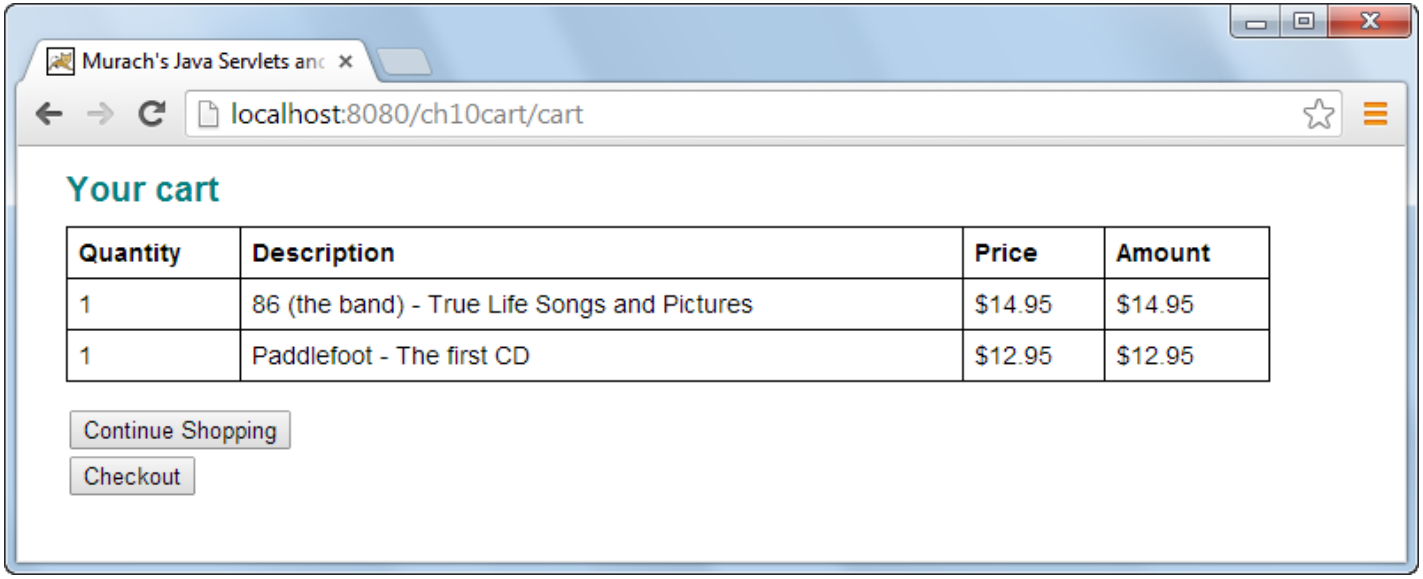
A custom tag that has attributes

- In the TLD file, the tag element can include the definitions for one or more attributes.
- In the TLD file, each attribute should include at least the name and required elements.
- In the tag class, declare a private instance variable for each attribute.
- In the tag class, define a set method for each attribute with the standard naming conventions.

How to use a tag that reiterates its body

```
<mma:cart>
  <tr>
    <td>${quantity}</td>
    <td>${productDescription}</td>
    <td>${productPrice}</td>
    <td>${total}</td>
  </tr>
</mma:cart>
```

A JSP that displays all items in a cart



A tag element for the TLD

```
<tag>
    <name>cart</name>
    <tag-class>murach.tags.CartTag</tag-class>
    <body-content>JSP</body-content>
</tag>
```

A tag class that reiterates the body

```
package murach.tags;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.util.*;
import java.io.IOException;

import murach.business.*;

public class CartTag extends BodyTagSupport {

    private ArrayList<LineItem> lineItems;
    private Iterator iterator;
    private LineItem item;
```

A tag class that reiterates the body (continued)

```
@Override
    public int doStartTag() {
        Cart cart = (Cart) pageContext.findAttribute("cart");
        lineItems = cart.getItems();
        if (lineItems.size() <= 0) {
            return SKIP_BODY;
        } else {
            return EVAL_BODY_BUFFERED;
        }
    }

    @Override
    public void doInitBody() throws JspException {
        iterator = lineItems.iterator();
        if (iterator.hasNext()) {
            item = (LineItem) iterator.next();
            this.setItemAttributes(item);
        }
    }
```

A tag class that reiterates the tag body (continued)

```
private void setItemAttributes(LineItem item) {
    Product p = item.getProduct();
    pageContext.setAttribute(
            "productCode", p.getCode());
    pageContext.setAttribute(
            "productDescription", p.getDescription());
    pageContext.setAttribute(
            "productPrice", p.getPriceCurrencyFormat());
    pageContext.setAttribute(
            "quantity", new Integer(item.getQuantity()));
    pageContext.setAttribute(
            "total", item.getTotalCurrencyFormat());
}
```

A tag class that reiterates the tag body (continued)

```
    @Override
    public int doAfterBody() throws JspException {
        try {
            if (iterator.hasNext()) {
                item = (LineItem) iterator.next();
                this.setItemAttributes(item);
                return EVAL_BODY_AGAIN;
            } else {
                JspWriter out = bodyContent.getEnclosingWriter();
                bodyContent.writeOut(out);
                return SKIP_BODY;
            }
        } catch (IOException ioe) {
            System.err.println("doAfterBody: " + ioe.getMessage());
            return SKIP_BODY;
        }
    }
}
```

A custom tag that reiterates its body

- To pass data to the tag class, store that data as a session attribute.
- To access a tag that has a body, the tag class must implement the BodyTag interface. The easiest way to do this is to extend the BodyTagSupport class.
- If the doStartTag method returns the EVAL_BODY_BUFFERED constant, the body of the tag is evaluated by calling the doInitBody method and the doAfterBody method.
- The doInitBody method sets the initial values for the first row of the body.
- If the doAfterBody method returns the EVAL_BODY_AGAIN constant, the doAfterBody method is called again.
- You can use the setAttribute method of the PageContext object to set any attributes that you need to access from the JSP tag.
- You can use the getEnclosingWriter and writeOut methods of the bodyContent object to write the body to the JSP.

An introduction to scripting variables

- If you're using version 2.0 or later of JSP, you can use EL with your custom tags to display attributes that were stored by the tag class.
- If you're using an older version of JSP, you need to use JSP expressions to display attributes that were stored by the tag class. In that case, you can create *scripting variables* to make it easier to display these attributes.

A custom JSP tag without scripting variables

```
<mma:cart>
    <tr valign="top">
        <td><%= pageContext.getAttribute("quantity") %></td>
        <td><%= pageContext.getAttribute("productDescription") %></td>
        <td><%= pageContext.getAttribute("productPrice") %></td>
        <td><%= pageContext.getAttribute("total") %></td>
    </tr>
</mma:cart>
```

A custom JSP tag with scripting variables

```
<mma:cart>
  <tr valign="top">
    <td><%= quantity %></td>
    <td><%= productDescription %></td>
    <td><%= productPrice %></td>
    <td><%= total %></td>
  </tr>
</mma:cart>
```

Code in the tag class that adds the scripting variables to the pageContext object

```
pageContext.setAttribute("productDescription", p.getDescription());
pageContext.setAttribute("productPrice", p.getPriceCurrencyFormat());
pageContext.setAttribute("quantity", new Integer(item.getQuantity()));
pageContext.setAttribute("total", item.getTotalCurrencyFormat());
```

A tag element in the TLD

```
<tag>
    <name>cart</name>
    <tag-class>tags.CartTag</tag-class>
    <tei-class>tags.CartTEI</tei-class>
    <body-content>JSP</body-content>
</tag>
```

Scripting variables

- The tag class must add the scripting variables to the pageContext object.
- The TEI class must define the scripting variables.
- The tag element in the TLD must specify both the tag class and the TEI class for the custom tag.

A TEI class that creates four scripting variables

```
package tags;

import javax.servlet.jsp.tagext.*;

public class CartTEI extends TagExtraInfo
{
    public VariableInfo[] getVariableInfo(TagData data)
    {
        return new VariableInfo[]
        {
           new VariableInfo(
             "productDescription", "String", true, VariableInfo.NESTED),
           new VariableInfo(
             "productPrice", "String", true, VariableInfo.NESTED),
           new VariableInfo(
             "quantity", "Integer", true, VariableInfo.NESTED),
           new VariableInfo(
             "total", "String", true, VariableInfo.NESTED),
        };
    }
}
```

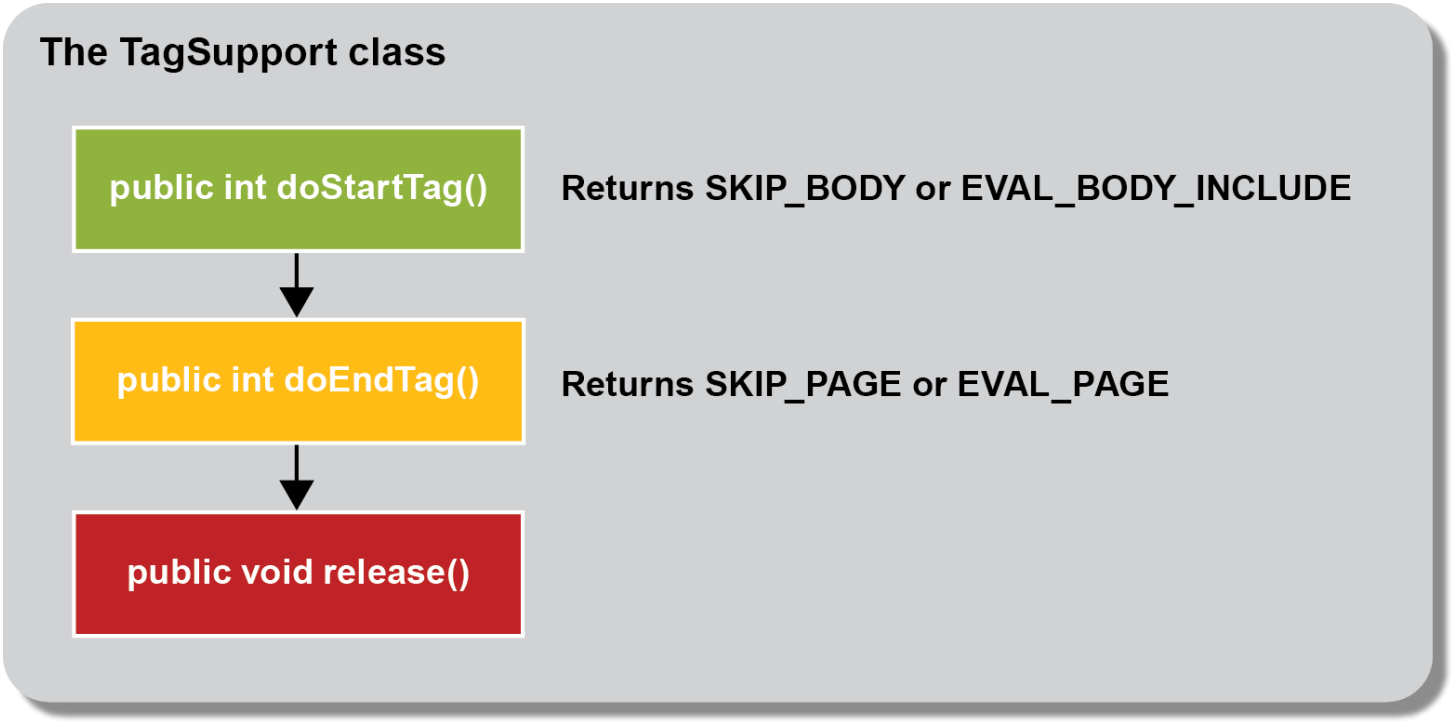
VariableInfo constants

Constant	The scope of the scripting variable is…
`AT_BEGIN`	From the start of the tag to the end of the JSP.
`AT_END`	From the end of the tag to the end of the JSP.
`NESTED`	From the start of the tag to the end of the tag.

TEI class

- To define scripting variables for a tag class, create a *tag extra information* (*TEI*) *class*. Store this class in the same location as the tag classes.
- To code a TEI class, extend the TagExtraInfo class. Then, override the getVariableInfo method to return an array of VariableInfo objects that define the scripting variables.
- For each scripting variable, create a VariableInfo object that provides this data: the name and data type of the variable, a true/false value that tells whether the variable needs to be declared, and the scope of the variable.
- For the data type of a scripting variable, specify a String object, any primitive data type, or any wrapper class for a primitive type.
- To specify whether the scripting variable needs to be declared, you can usually specify a true value to indicate that the variable is new and should be declared.

Common methods and fields of the TagSupport class



Methods and fields of the TagSupport class

- The doStartTag method is the first method that's called for a custom tag. Typically, this method contains the statements that perform the processing for the tag.
- If a tag doesn't have a body, the doStartTag method should return the SKIP_BODY field. That way, the body of the tag won't be displayed.
- If a tag has a body, the doStartTag method should return the EVAL_BODY_INCLUDE field. That way, the body of the tag is displayed.
- To display the rest of the JSP after the custom tag, the doEndTag method should return the EVAL_PAGE field.
- To not display the rest of the JSP after the custom tag, the doEndTag method should return the SKIP_PAGE field.
- If you need to execute any statements that release any system resources that the tag is using, you can code a release method.

The pageContext object in the TagSupport class

```
protected PageContext pageContext
```

Common methods of the PageContext class

Method	Description
`getOut()`	Returns JspWriter object from JSP.
`getRequest()`	Returns request object from JSP.
`getResponse()`	Returns response object from JSP.
`setAttribute(String name, Object o)`	Sets named attribute with page scope to the value.
`setAttribute(String name, Object o, int scope)`	Sets named attribute with specified scope to the value.
`getAttribute(String name)`	Searches page scope for an attribute with specified name.
`getAttribute(String name, int scope)`	Searches specified scope for an attribute with specified name.
`findAttribute(String name)`	Searches page, request, session, and application scopes in that sequence for specified attribute.

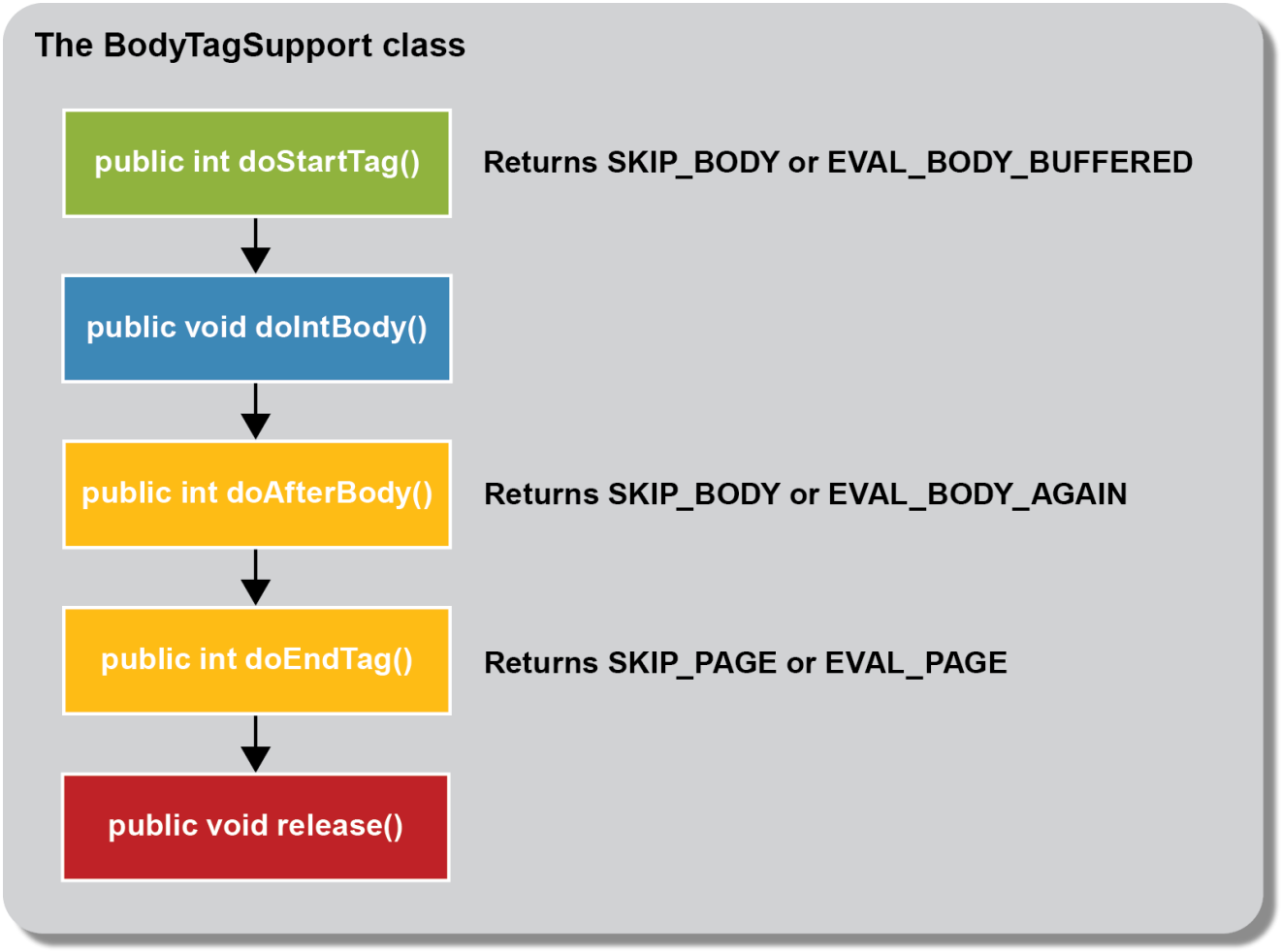
The fields of the PageContext class for setting scope

```
PAGE_SCOPE
REQUEST_SCOPE
SESSION_SCOPE
APPLICATION_SCOPE
```

Methods and fields of the PageContext class

- Use the pageContext object to set and get JSP objects and attributes.
- For more information about the PageContext class, look in the javax.servlet.jsp package of the Java EE API documentation.

Methods and fields of the BodyTagSupport class



Methods and fields of the BodyTagSupport class

- If you want to perform some initial processing for a tag, override the doStartTag method of the BodyTagSupport class.
- If the doStartTag method returns the EVAL_BODY_BUFFERED field, the doInitBody and doAfterBody methods are called to display the body of the tag.
- The doInitBody method should contain all of the initialization statements that are needed for the first evaluation of the body.
- The doAfterBody method should contain all of the statements that are needed for additional evaluations of the body.
- If the doAfterBody method returns the EVAL_BODY_AGAIN field, the body is added to the bodyContent object and the doAfterBody method is called again.
- If the doAfterBody method returns the SKIP_BODY field, the processing for the tag is finished and the body is skipped.

The bodyContent object in the BodyTagSupport class

```
protected BodyContent bodyContent
```

Common methods of the BodyContent class

Method	Description
`clearBody()`	Clears the body.
`getEnclosingWriter()`	Returns the JspWriter object for the body.
`getString()`	Returns the body as a String object.
`writeOut(Writer out)`	Writes the body to the specified out stream.

Methods and fields of the BodyContent class

- The bodyContent object stores the body of the tag before it is written to the JSP.
- To display the body in the JSP, use the getEnclosingWriter and writeOut methods of the BodyContent class.
- For more information about the BodyContent class, look in the javax.servlet.jsp.tagext package in the Java EE API documentation.