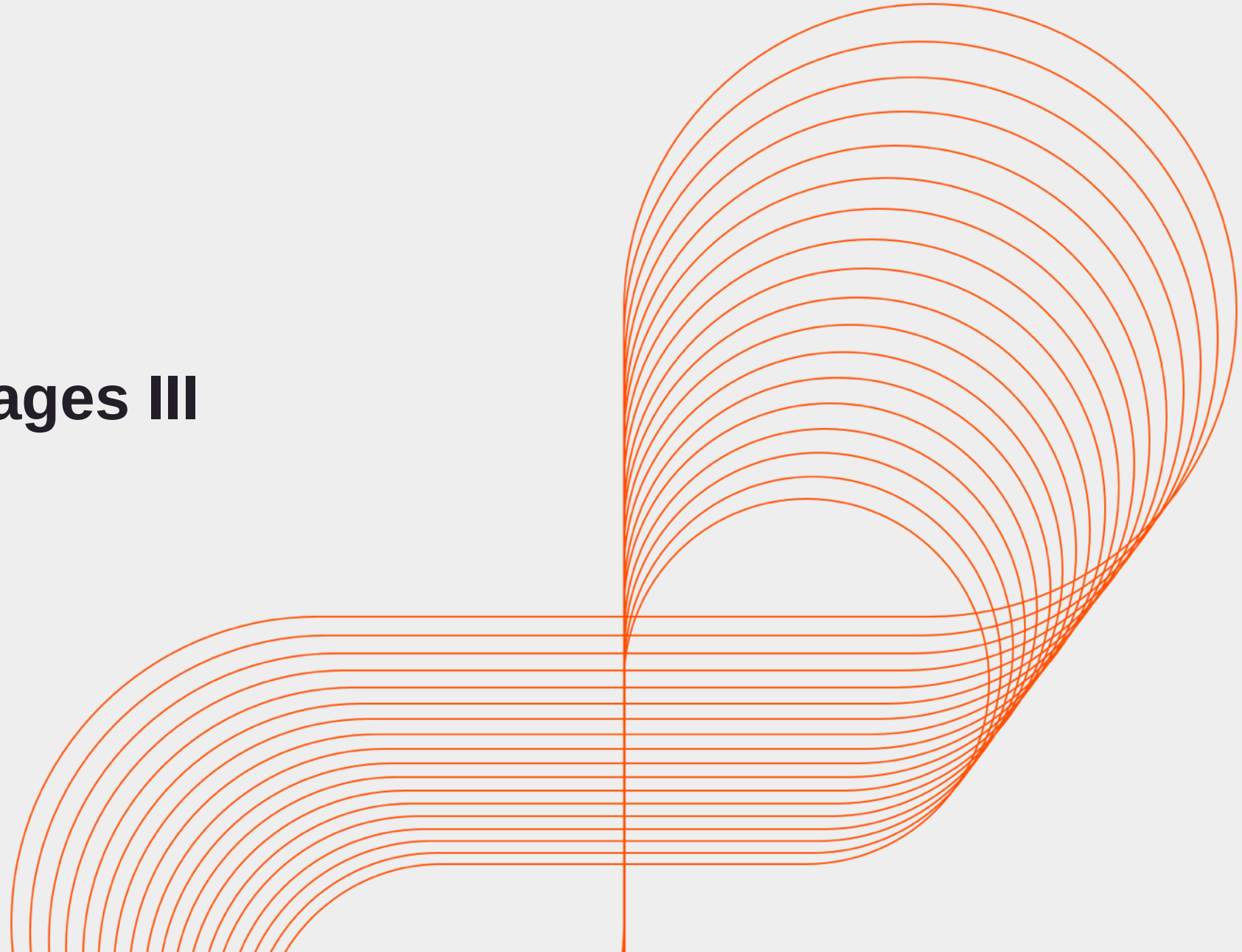
Persistent

Java Server Pages III

Customizing your pages



Agenda

- Custom tags/actions
  - Classic tag handler
  - Simple tag handler
- EL functions

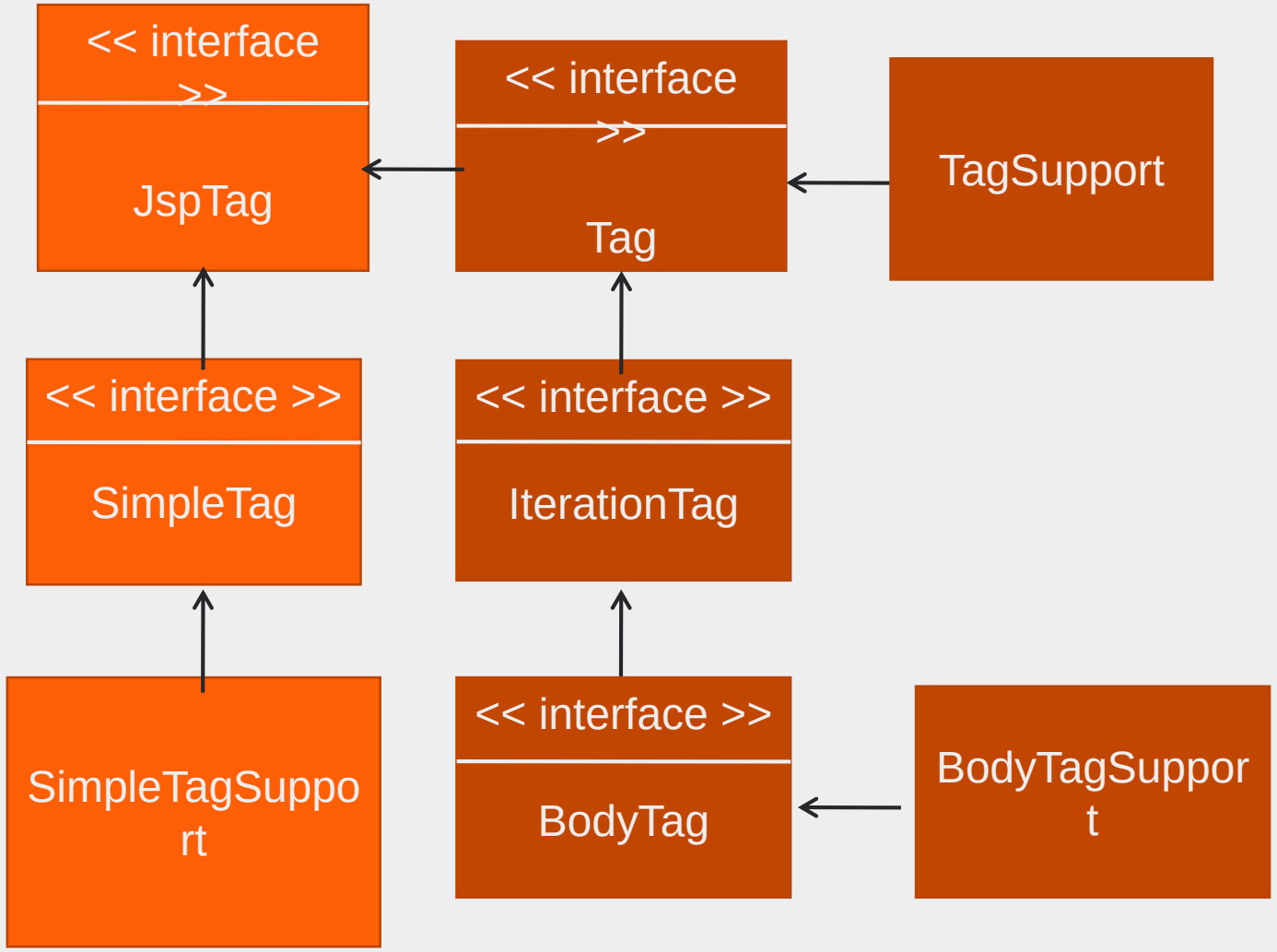
Need for custom actions

- Make it easier for content developers to retrieve and present the required information through simple *html*-like tags instead of writing complex logic with scriptlets.

- Help improve the separation between business logic and presentation logic.

- Custom tags are reusable and can thus help save significant development and testing time.

Custom tags API overview



Classic tag handlers - Example

- **Current system**
  - A web application’s user registration form has a combination of three drop down list boxes for entering a prospective user’s date of birth.
- **Problem/Issues with current approach**
  - The UI designers are required to write substantial JSTL for populating the three combo boxes with relevant values i.e. 1-31 for day, Jan-Dec for month and 1900-1996 for year. Other than that they end-up replicating this in other pages wherever the same fields are required.
- **Requirement**
  - The UI designers have asked the programmers to come up with a flexible approach which makes presenting a date of birth field on a web page convenient.
- **Solution**
  - The programmers have decided to create a bodyless custom tag with attributes

Classic tag handlers – Step 1 – The tag class

```
// package statement

// necessary imports

public class DateOfBirthTag extends TagSupport {

   private String dayFieldName,
                          monthFieldName,  yearFieldName;

   private int startingYear, endingYear;

   private JspWriter out;

   private enum Months { Jan, Feb, Mar, Apr, May,
            Jun, Jul, Aug, Sep, Oct,
    Nov, Dec };

   public DateOfBirthTag() { /*  does nothing */ }

   /* setters and getters for all tag attributes */
```

Create a class which inherits from `javax.servlet.jsp.tagext.TagSupport`

Tag attributes as data members of the tag class

Some additional useful data members. Not attributes.

Set and get methods for each attribute identified above.

Classic tag handlers – Step 1 contd..

```
public int doStartTag() throws JspException {
    out = pageContext.getOut();
        PrintWriter prn = new PrintWriter(out);
            prn.printf("<select name=\"%1$s\">", getDayFieldName());
          for(int _day = 1 ; _day <= 31 ; _day++)
        prn.printf("<option value=\"%1$d\">%1$d</option>", _day);
        prn.print("</select>");
    prn.printf("<select name=\"%1$s\">", getMonthFieldName());
for(Months _month : Months.values())
prn.printf("<option value=\"%1$d\">%2$s</option>",
                    _month.ordinal() + 1, _month);
prn.print("</select>");
prn.printf("<select name=\"%1$s\">", getYearFieldName());
for(int _year = getStartingYear() ; _year <= getEndingYear() ; _year++)
prn.printf("<option value=\"%1$d\">%1$d</option>", _year);
prn.print("</select>");
```

Override the *doStartTag( )* method

Fetch a reference to the *JspWriter* through the inherited *pageContext* object

Write out the html required to produce the combo boxes

Return an appropriate integral constant defined in the `javax.servlet.jsp.tagext.Tag` interface

Override the *doEndTag( )* method

Continued

```
return SKIP_BODY;
}
public int doEndTag() throws JspException { return EVAL_PAGE; }
```

Return an appropriate integral constant defined in the `javax.servlet.jsp.tagext.Tag` interface

Override the *doEndTag( )* method

Return an appropriate integral constant defined in the `javax.servlet.jsp.tagext.Tag` interface

Classic tag handlers – Step 2 – The TLD

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

    <taglib 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"

            version="2.0">

      <tlib-version>1.2</tlib-version>

      <uri>http://sample-web-app.co.in/tags</uri>
```

Current version of your tag library

Unique identifier of your tag library in the form of a URI. This will be used in the *taglib* directive

Continued

```
<tag>
    <name>date-of-birth</name>
    <description>A classic custom html select tag for date of birth</description>
    <tag-class>com.sample.tag.DateOfBirthTag</tag-class>
    <body-content>empty</body-content>
    <attribute>
        <name>dayFieldName</name>
        <required>true</required>
        <rtexprvalue>false</rtexprvalue>
    </attribute>
```

Name of your tag.

Package qualified name of your tag class.

Indicates whether the tag has a body or not

Attribute details

Persistent

Classic tag handlers – Step 2 contd..

```
<attribute>
        <name>monthFieldName</name>
        <required>true</required>
        <rtexprvalue>false</rtexprvalue>
</attribute>
<attribute>
        <name>yearFieldName</name>
        <required>true</required>
        <rtexprvalue>false</rtexprvalue>
</attribute>
<attribute>
```

Continued

```
            <name>startingYear</name>
            <required>true</required>
            <rtexprvalue>true</rtexprvalue>
            <type>java.lang.Integer</type>
        </attribute>
        <attribute>
            <name>endingYear</name>
            <required>true</required>
            <rtexprvalue>true</rtexprvalue>
            <type>java.lang.Integer</type>
        </attribute>
    </tag>
</taglib>
```

Attribute details of this tag:

- *name* - Name of the attribute. Must match the data member in the tag class.
- *required* – Indicates whether specifying a value for this attribute is mandatory. One of either true or false.
- *rtexprvalue* – Indicates whether the right hand side i.e. value of this attribute can be an EL expression.
- *type* – Indicates how the value of the attribute should be treated when it is assigned to the tag class data member. By default string.

Classic tag handlers – Step 3 – The JSP

```
<%@page              language="java“ contentType="text/html; charset=ISO-8859-1"
        pageEncoding="ISO-8859-1"  %>
<%@taglib prefix="myTag" uri="http://sample-web-app.co.in/tags" %>
<html>
   <head>
        <title>A sample JSP</title>
   </head>
   <body>
<myTag:date-of-birth monthFieldName="month"
   yearFieldName="year"
Day FieldName="day" endingYear="1996"
   startingYear="1900"/>
   </body>
</html>
```

Taglib directive for your custom tag library. Prefix can be anything but URI must match the one given in the tld.

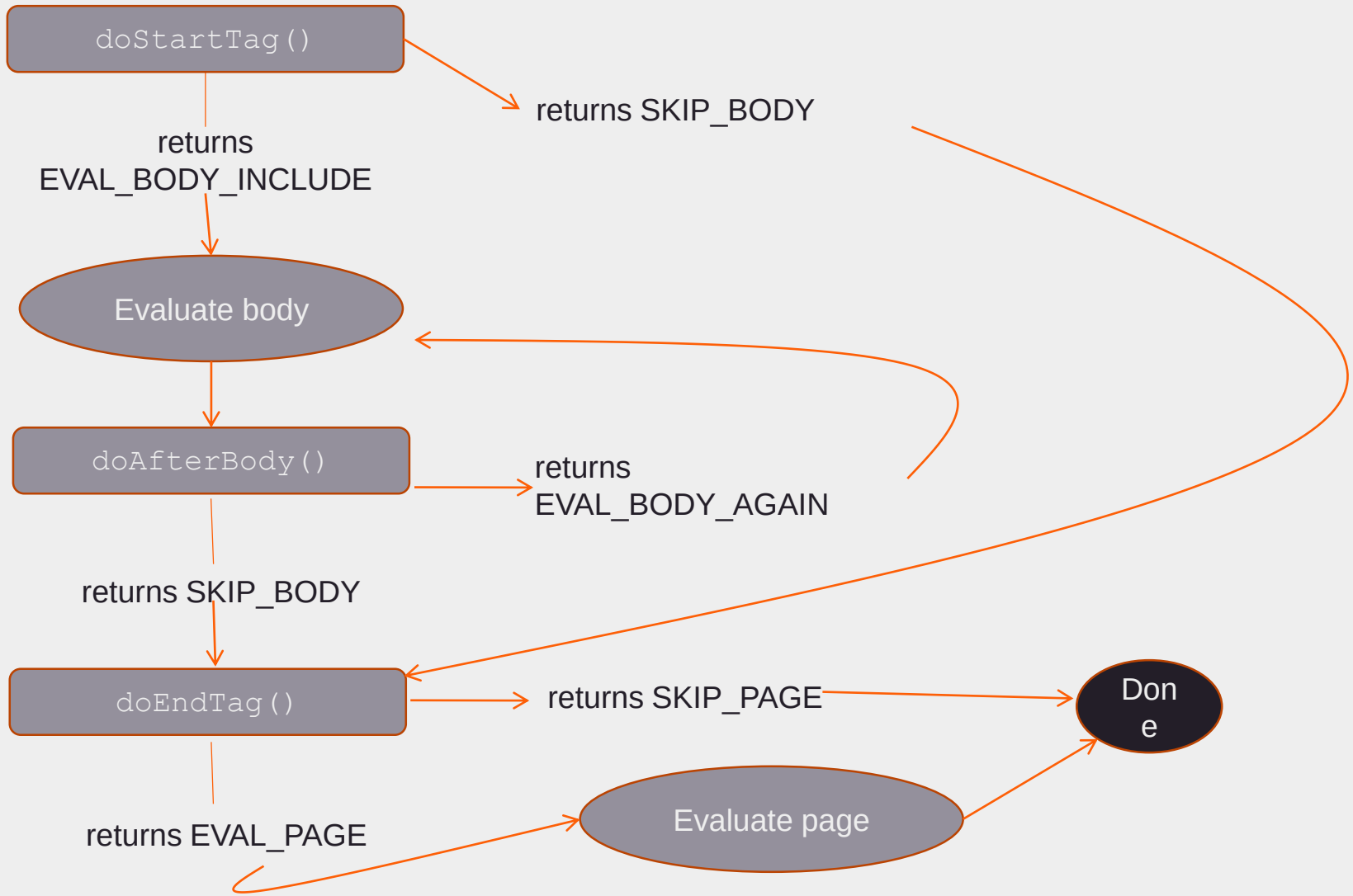
Your custom tag with attribute values

Persistent

Classic tags – How stuff works ?

- The JSP translator when it encounters the custom tag looks-up for a tag descriptor which contains various details of the tag like its tag class, attributes etc.
- This look-up is done on the basis of URI and the tag's name.
- It then generates Java code which
  - instantiates the tag class
  - invokes the *setPageContext(PageContext)* method on the object. This gives the tag handler reference to the page context object.
  - invokes the *setParent(Tag)* method.
  - invokes setters on it for each attribute thereby passing values specified in the tag.
  - invokes the *doStartTag( )* method on the object
  - evaluates the tag's body, if the tag is declared to have one.
  - invokes the *doAfterBody( )* method on the object.
  - invokes the *doEndTag( )* method on the object.

Classic tags – Understanding the workflow



Persistent

Classic tag handlers – Tag with a body

- **Current system**
  - The page creators of a web application often add temporary jstl, scriptlets and other stuff to debug a page whenever a issue is reported.

- **Problem/Issues with current approach**
  - The page creators have to remove all extra debugging “code” before the pages are deployed. If an issue recurs or a new issue is reported for the same page, the “code” needs to be rewritten.

- **Requirement**
  - The page creators are looking for a flexible approach wherein they can simply enclose all their “debugging code” within a tag and then turn the debug mode on or off on a page by page basis.

- **Solution**
  - The programmers have decided to create a custom tag with body which will look for a request parameter to either process or bypass the tag.

Classic tags with body – Step 1 – The tag class

```
// package statement

// necessary imports

public class DebugModeTag extends BodyTagSupport {

    private JspWriter out;

    public DebugModeTag() { /* does nothing */ }

    public int doStartTag() throws JspException {

            out = pageContext.getOut();

            try {

                    if(pageContext.getRequest().getParameter("debug") != null) {

                            out.write("<div style=\"background-color: graytext;\">");

                            return EVAL_BODY_INCLUDE;

                    } else return SKIP_BODY;
```

Create a class which inherits from `javax.servlet.jsp.tagext.BodyTagSupport`

Override the `doStartTag()` method

Return `EVAL_BODY_INCLUDE` if the body is to evaluated i.e. the `doAfterBody()` method is to be invoked or return `SKIP_BODY` if `doEndTag()` is to be invoked.

Continued

```
} catch (IOException e) { throw new JspException("Start tag processing failed", e); }
    }
    public int doAfterBody() throws JspException {
            try {
                    out.write("</div>");
                    return SKIP_BODY;
            } catch (IOException e) { throw new JspException("Start tag processing failed", e); }
    }

    public int doEndTag()  { return EVAL_PAGE; }
}
```

Override the `doAfterBody()` method

Return either `EVAL_BODY_AGAIN` to evaluate the body again and call this method or `SKIP_BOD Y` to invoke the `doEndTag()` method.

Classic tags with body – Step 2 – The TLD

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<taglib 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" version="2.0">
    <tlib-version>1.2</tlib-version>
    <uri>http://sample-web-app.co.in/tags</uri>
    <tag>
        <name>debug-mode</name>
        <description>A utility custom tag for debugging</description>
        <tag-class>com.sample.tag.DebugModeTag</tag-class>
        <body-content>jsp</body-content>
</tag>
</taglib>
```

body are:
`scriptless` - Body can contain anything (other tags, el and static text) other than JSP scripting elements (scriptlet, declaration, expression)
`tagdependent` - Evaluating the body content is entirely upto the tag handler class.

Persistent

Classic tags with body – Step 3 – The JSP

```
<%@taglib prefix="myTag" uri="http://sample-web-app.co.in/tags" %>
<myTag:debug-mode>
    <%
            // scriptlet with some Java code
    %>
    <br/>
    <%=application.getInitParameter("WEBMASTER_EMAIL") %>
    <br/>
</myTag:debug-mode>
<myTag:debug-mode>
    // some jstl tags here
    // some el here
</myTag:debug-mode>
```

- The page must be accessed with a request parameter named debug for the tag's body to be processed.

Simple tag handlers - Example

- Re-implementing the same requirement covered earlier for bodyless classic tag with a bodyless simple tag.

Simple tag handler – End to end

```
// package statement

// necessary imports

public class DateOfBirthTag extends SimpleTagSupport {

    // data members as declared in the earlier sample

    public DateOfBirthTag() { /* does nothing */ }

    // setters and getters for each attribute

  public void doTag() throws JspException, IOException {

          out = getJspContext().getOut();

            // exactly same code as covered earlier

    }

}
```

Create a class which inherits from `javax.servlet.jsp.tagext.SimpleTagSupport`

Override the `doTag()` method. This is the only method that needs to be overridden.

Simple tags have access to a JSP context (`javax.servlet.jsp.JspContext`) rather than page context. JspContext is the super class of PageContext

- There will be no change in the tag descriptor created earlier.
- The tag will be used in a JSP in the same way as was done earlier.

Lifecycle of a simple tag handler

- Load the tag class
- Instantiate the tag class
- Invoke the *setJspContext(JspContext)* method on the object. This gives the tag handler reference to the page context object (sub-class of jsp context).
- Invoke the *setParent(JspTag)* method.
- Invoke setters on it for each attribute thereby passing values specified in the tag.
- If the tag is declared to have a body, invoke the *setJspBody(JspFragment)* method.
- Invoke the *doTag( )* method on the object

Simple tags – Tag with a body

- Re-implementing the same requirement covered earlier for a classic tag with a body as a simple tag with body.

Simple tag with body – Step 1 – The tag class

```
// package statement
// necessary imports
public class DebugModeTag extends SimpleTagSupport {
    private JspWriter out;
    public DebugModeTag() { /* does nothing */  }
    public void doTag() throws JspException, IOException {
        out = getJspContext().getOut();
        PageContext pageContext;
        pageContext = (PageContext) getJspContext();
        try {
            if(pageContext.getRequest().getParameter("debug") != null) {
                out.write("<div style=\"background-color: graytext;\">");
        getJspBody().invoke(null);
                out.write("</div>");
    }
```

Create a class which inherits from `javax.servlet.jsp.tagext.SimpleTagSupport`

Override the `doTag()` method

Do the required processing

Must call the `invoke()` method on an object of `JspFragment` for evaluating the body. The `getJspBody()` method returns a reference to an object of `JspFragment`.

continued

```
} catch (IOException e) {
                        throw new JspException("Tag processing failed", e);
            }
    }
}
```

Simple tag with body – Step 2 & 3 – The tld and the jsp

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
    <taglib …….>
    <tlib-version>1.2</tlib-version>
    <uri>http://sample-web-app.co.in/tags</uri>
    <tag>
            <name>debug-mode</name>
            <description>A utility custom tag for debugging</description>
            <tag-class>com.sample.tag.DebugModeTag</tag-class>
            <body-content>scriptless</body-content>
    </tag>
</taglib>
```

The only possible values for `<body-content>` in a tag with body are `scriptless` and `tagdependent`. `jsp` is not allowed .

- The tag will be used as was done earlier but page creators must be careful not to write any scriptlets in the body.

Tag files

- A file which contains a fragment of JSP (other custom tags, standard actions, jstl, el, directives and scriptlets) that is reusable as a custom tag.

- Allow page developers create custom tags without having to write any Java code.

- Don’t need a TLD unless packed into a jar and deployed.

Tag file - Example

- Re-implementing the date of birth tag as a tag file.
- Pre-requisite
  - Create a folder named tags under WEB-INF. Your tag files will be placed in this directory.
  - Your tag files must have an extension of .tag

The tag file – Step 1

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

Taglib directive for using other tag libraries in a tag file.

```
<%@tag body-content="empty" %>
```

`Tag` directive with `body-content` attribute. This directive has attributes similar to the `page` directive in a JSP

```
<%@attribute name="dayFieldName" required="true" rtexprvalue="false" %>
<%@attribute name="monthFieldName" required="true" rtexprvalue="false" %>
<%@attribute name="yearFieldName" required="true" rtexprvalue="false" %>
<%@attribute name="startingYear" required="true" rtexprvalue="true"
                type="java.lang.Integer"%>
<%@attribute name="endingYear" required="true" rtexprvalue="true"
                type="java.lang.Integer"%>
<select name="${dayFieldName}">
   <c:forEach begin="1" end="31" var="_day">
            <option value="${_day}">${_day}</option>
   </c:forEach>
```

`Attribute` directive which declares the various attributes of this tag.

Continued

```
</select>
<select name="${monthFieldName}">
    <c:forTokens  items="Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec"
                            delims="," var="_month" varStatus="_status">
            <option value="${_status.count}">${_month}</option>
    </c:forTokens>
</select>
<select name="${yearFieldName}">
    <c:forEach begin="${startingYear}" end="${endingYear}" var="_year">
            <option value="${_year}">${_year}</option>
    </c:forEach>
</select>
```

Tag file – Step 2 – The JSP

```
<%@page language="java" contentType="text/html; charset=ISO-8859-1"
        pageEncoding="ISO-8859-1"  %>
<%@taglib prefix="myTag" tagdir="/WEB-INF/tags" %>
<html>
    <head>
        <title>A sample JSP</title>
    </head>
    <body>
        <myTag:date-of-birth monthFieldName="month"
                             yearFieldName="year"
    dayFieldName="day"
                             endingYear="1996"
                             startingYear="1900"/>
    </body>
</html>
```

Taglib directive with `tagdir` attribute instead of `uri`

Tag file with a body - Example

- Re-implementing the debug mode tag as a tag file

The tag file – Step 1 and 2

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

<%@tag body-content="scriptless" %>

<c:if test="${param.debug ne null}">

        <div style="background-color: graytext;">

                <jsp:doBody/>

        </div>

</c:if>
```

Must be invoked for processing the body

- The JSP will be similar to one created earlier
- Key points
  - A tag file can contain scriptlets, declarations and expressions but the body of the tag (i.e. on the JSP) cannot have these elements.
  - A tag file has access to all those implicit objects that a JSP has except page context. It has a JspContext instead.

EL functions

- Application specific utility methods which can be invoked from an EL

- Helps avoid scripting for calling methods

EL functions – Step 1 – The class

- A simple Java class with methods. Methods must be public and static.

```
public class Methods {
        /**
         * @return either true or false
         * @param year - an integral year
         * @description A method to determine whether a year is leap or not
         */
        public static boolean isLeap(int year)
        {
                if(year > 0) {
                        if(year % 100 == 0) {
                                if(year % 400 == 0)  return true;
```

Continued

```
else                    return false;
                        } else {
                                if(year % 4 == 0)     return true;
                                else                        return false;
                        }
                } else return false;
        }
}
```

EL functions – Step 2 – The tld

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<taglib ……>
        <tlib-version>1.2</tlib-version>
        <uri>http://sample-web-app.co.in/methods</uri>
        <function>
                <name>isLeap</name>
                <function-class>
                        com.sample.util.Methods
                </function-class>
                <function-signature>
                        boolean isLeap(int)
                </function-signature>
        </function>
</taglib>
```

Name of the function. This need not match name of the method in the class

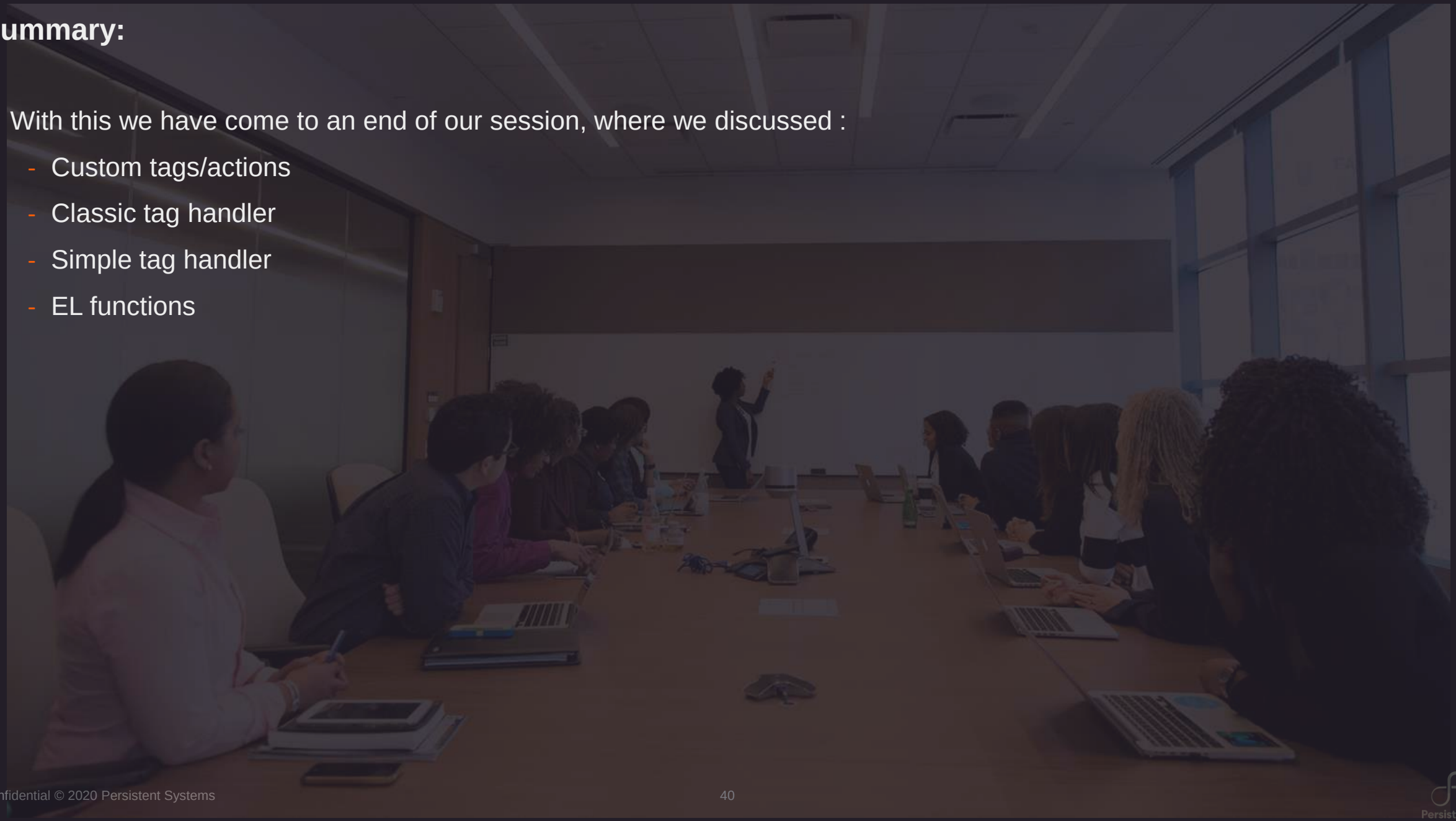
Package qualified name of the class which contains the methods.

Actual signature of the method as given in the class.

EL functions – Step 3 – The JSP

```
<%@ page language="java"
        contentType="text/html;
        charset=ISO-8859-1" pageEncoding="ISO-8859-1"  %>
<%@taglib        prefix="myMethod"
                 uri="http://sample-web-app.co.in/methods" %>
<html>
        <head>
                 <title>A sample JSP</title>
        </head>
        <body>
                 ${myMethod:isLeap(2000)}
                 <br/>
        </body>
</html>
```

Summary:



- With this we have come to an end of our session, where we discussed :
  - Custom tags/actions
  - Classic tag handler
  - Simple tag handler
  - EL functions

Appendix

Thank You

Persistent

Thank you