



# Java Technologies

## Custom Tag Libraries

# The Context

- JavaServer Pages offer a standard solution to **create web content** dynamically using **JSP elements**: JSP tags, scriptlets, etc.
- JSP are used to create **the view** of the application (the presentation layer)
- Presentation → Designer, ~~not Programmer~~
- **How can we generate dynamic content without writing Java code?**
  - The first step: **JSP Standard Actions**  
(jsp:useBean, jsp:forward, jsp:include, etc)

# The Concept

We need a component that:

- **Encapsulates** various types of non-standard dynamic functionalities such as:
  - generating an HTML table with available products
  - extracting some data from an XML document, etc.
- Can be used inside JSP in a **similar** manner **to the standard actions**
  - the programmer writes the functionality
  - the designer accesses the functionality in a declarative fashion (using a tag)
- Promotes code **reusability** (libraries)

# Separation of Concerns (Soc)

- A design principle for **separating a system into distinct sections**, such that:
  - each section addresses a separate concern
  - the overlapping should be minimal
- Edsger W. Dijkstra: "*On the role of scientific thought*" (1974)
  - "focusing one's attention upon some aspect"
  - "the only available technique for effective ordering of one's thoughts"

# Custom Tags in JSP

- A custom tag is a **user-defined JSP element**
- The object that implements a custom tag is called a **tag handler**
  - *class* (programmatically)
  - *tag file* (JSP fragment)
- A tag is invoked in a JSP file using XML syntax:

```
<prefix:tagName>  
    Body  
</prefix:tagName>
```

- When a JSP is translated into a servlet, the tag is converted to operations on the tag handler.

# Custom Tag Features

- Customized by means of attributes passed from the calling page
- Pass variables back to the calling page
- Access all the objects available to JSP pages
- Communicate with each other
- Be nested within one another and communicate by means of private variables
- Distributed in a tag library

# Creating a Tag Handler

```
public class HelloTagHandler extends SimpleTagSupport {

    /**
     * Called by the container to invoke this tag. The
     * implementation of this method is provided by the tag
     * library developer, and handles all tag processing
     */

    @Override
    public void doTag() throws JspException, IOException {

        // Create dynamic content
        JspWriter out = getJspContext().getOut();
        out.print("Hello World from Infoiasi!");
    }

}
```

# The Tag Library Descriptor (TLD)

Defines a mapping between tag handlers (classes) and tag names

**<taglib>**

<tlib-version>1.0</tlib-version>

<short-name>mylibrary</short-name>

**<uri>/WEB-INF/tlds/mylibrary</uri>**

<tag>

**<name>hello</name>**

**<tag-class>HelloTagHandler</tag-class>**

<description> Displays the Hello World message (again) </description>

<body-content>**empty**</body-content>

</tag>

**</taglib>**



# Using the Custom Tag

```
<b>Somewhere, inside a JSP</b>
```

```
<p>
```

```
<%@ taglib uri="/WEB-INF/tlds/mylibrary"  
      prefix="say" %>
```

Use the *taglib directive* to  
specify the tag library

```
<say:hello/>
```

Use the custom tag

# Custom Tag API

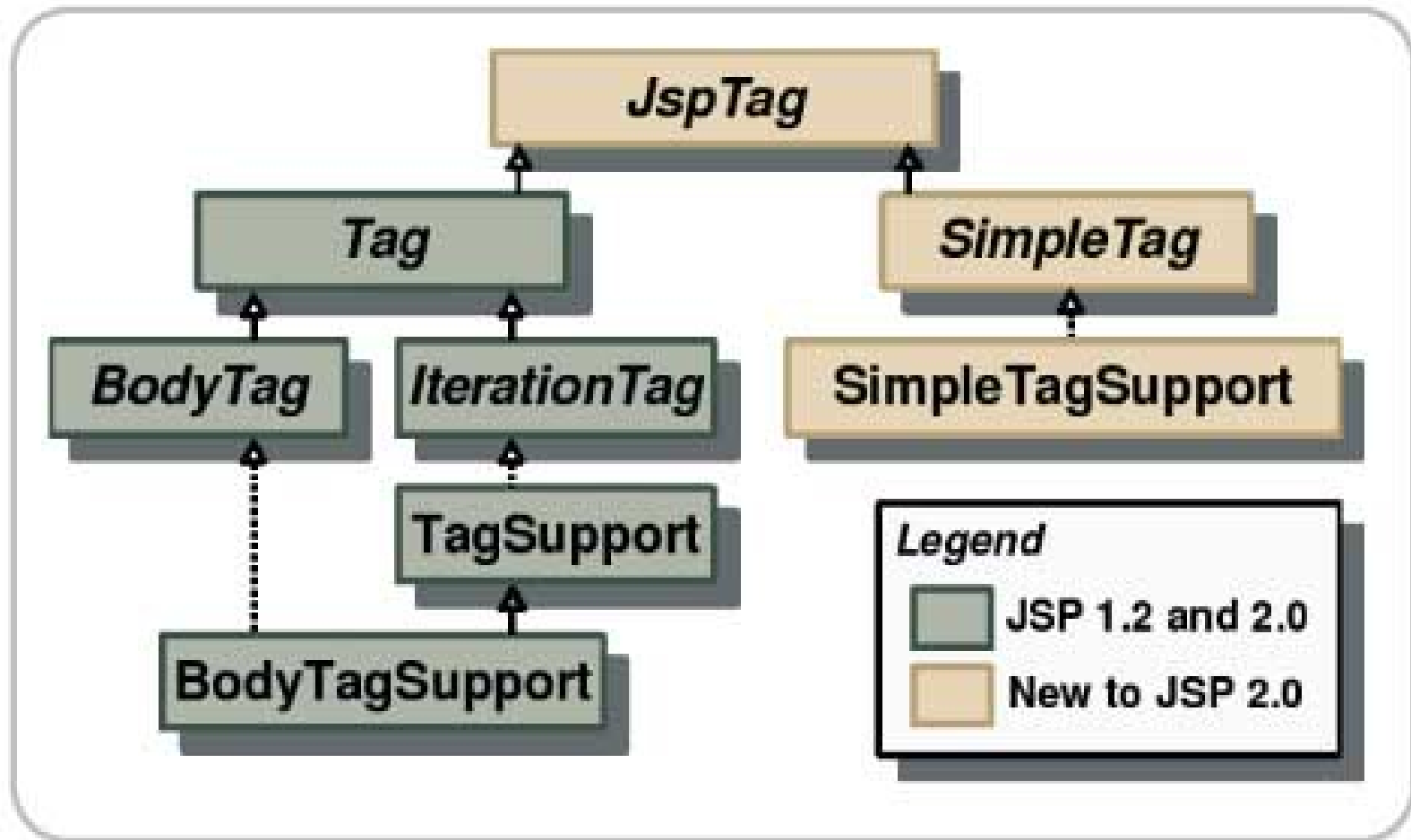


Figure 1 Tag extension class hierarchy

Simple tag handlers can be used only for tags that do not use scripting elements in attribute values or the tag body. Classic tag handlers must be used if scripting elements are required.


# Using Attributes in Custom Tags

- Attributes customize the behavior of a tag
- `<say:hello message="Hello World" />`
- Declaring the attribute in the TLD:

```
<tag>
  ...
  <attribute>
    <name>message</name>
    <required>true</required>
    <rtexprvalue>false</rtexprvalue>
    <type>String</type>
  </attribute>
</tag>
```

# Using attributes in the Handler

```
public class HelloTagHandler extends SimpleTagSupport {  
  
    private String message;  
  
    public String getMessage() {  
        return message;  
    }  
  
    public void setMessage(String message) {  
        this.message = message;  
    }  
  
    @Override  
    public void doTag() throws JspException, IOException {  
        JspWriter out = getJspContext().getOut();  
        out.print(message + " from Infoiasi!");  
    }  
}
```



```
<attribute>  
    <name> message </name>  
    ...  
</attribute>
```

When the JSP container creates the tag handler instance, it uses *reflection* to set the properties corresponding to the attributes.

# Dynamic Attributes

- Dynamic attributes not specified in the definition of the tag, usually because their names are not known at development time.

`<say:hello msg1="Hello" msg2="Ciao" msg3="Salut"/>`

- In the TLD file

```
<dynamic-attributes>true</dynamic-attributes>
```

- In the tag handler:

```
private void Map<String, Object> attributes;  
public void setDynamicAttribute(  
    String uri, String name, Object value) throws JspException {  
    attributes.put(name, value);  
}
```

# Tags with Bodies

- A simple tag can contain custom and core tags, HTML text, and tag-dependent body content between the start tag and the end tag.
- In the TLD file:

```
<body-content> empty | tagdependent | scriptless </body-content>
```

**tagdependent** = The body of the tag is interpreted by the tag implementation itself, and is most likely in a different language, for example, embedded SQL statements.

```
<sql:query> select name from products </sql:query>
```

**scriptless** = the body accepts only static text, EL expressions, and custom tags. No scripting elements are allowed.

```
<some:tag>  
  <h1> Some html </h1>  
  <say:hello />  
</some:tag>
```

# Evaluating the Body of a Tag

```
public void doTag() throws JspException {
    JspWriter out = getJspContext().getOut();
    try {
        // Insert code to write html before writing the body content.
        // e.g.:
        // out.println("<strong>" + attribute_1 + "</strong>");
        // out.println("    <blockquote>");

        JspFragment f = getJspBody();
        if (f != null) {
            f.invoke(out);
            //Executes the fragment and directs all output to the given
            //Writer, or the JspWriter returned by the getOut() method of
            //the JspContext associated with the fragment if out is null.
        }

        // TODO: insert code to write html after writing the body content.
        // e.g.:
        // out.println("    </blockquote>");

    } catch (java.io.IOException ex) {
        throw new JspException("Error in the handler tag", ex);
    }
}
```

# Example

```
public void doTag() throws JspException {
    JspWriter out = getJspContext().getOut();
    try {
        out.println("<strong>");
        out.println("Here comes the body of the tag in uppercase:<br/>");

        JspFragment f = getJspBody();
        if (f != null) {
            // get the body and process it
            StringWriter sw = new StringWriter();
            f.invoke(sw);
            String result = sw.toString().toUpperCase();

            //send the result to the output of the page
            out.println(result);
        }

        out.println("</strong>");

    } catch (java.io.IOException ex) {
        throw new JspException("Error in the handler tag", ex);
    }
}
```



# Nested Tags

- Tags can be imbricated:

```
<tt:outerTag>  
    <tt:innerTag />  
</tt:outerTag>
```

- Example

```
<c:if test="<%= expression %>">  
    <c:then>  
        JSP fragment included if the expression is true  
    </c:then>  
  
    <c:else>  
        JSP fragment included if the expression is false  
    </c:else>  
</c:if>
```

# Communication between Tags

- In the case of nested tags

```
public class IfTag extends SimpleTagSupport {
    boolean condition;
    ...
}
public class IfThenTag extends SimpleTagSupport {
    public void doTag() throws JspTagException {
        IfTag parent =(IfTag)findAncestorWithClass(this, IfTag.class);
        if (parent == null) {
            throw new JspTagException("then not inside if");
        }
        ...
    }
}
```

- In the general case: using *attributes* defined at *page scope*

```
PageContext context = (PageContext)getJspContext();
context.setAttribute("key", value);
```

# Tag Files

- A tag file is a source file that contains a fragment of JSP code that is reusable as a custom tag.
- Tag files allow you to create custom tags using JSP syntax. Just as a JSP page gets translated into a servlet class and then compiled, a tag file gets translated into a tag handler and then compiled.
- The recommended file extension for a tag file is **.tag** (.tagf for a fragment of a tag file).

# Example of a Tag File

- Creating a tag file

**/WEB-INF/tags/hello.tag**

```
<%@ attribute name="message" required="true" %>
<h1>
    <%= message %> from Infoiasi!
</h1>
```

- Using a tag file in a JSP

```
<%@ taglib tagdir="/WEB-INF/tags" prefix="say" %>
<html>
    <say:hello message = "Salut"/>
</html>
```

# Using the Body Content

```
<jsp:doBody var="content"/>
```

```
<%  
    String bc = (String) jspContext.getAttribute("content");  
    bc = bc.toUpperCase();  
%>
```

```
<h2>  
    Here comes the body: <%= bc %> !  
</h2>
```



How to get rid of  
the Java code?