



# Tehnologii Java

*Curs -*

Cristian Frăsinaru

`acf@infoiasi.ro`

Facultatea de Informatică

**Universitatea "Al. I. Cuza" Iași**



# Taguri JSP proprii

# Cuprins

---

- Ce sunt tagurile proprii ?
- Avantaje
- Crearea și folosirea
- Tipuri de taguri
- Cooperarea între taguri
- Fișiere de taguri



# Introducere



# Ce sunt tagurile proprii ?

Tagurile proprii sunt elemente JSP ce **încapsulează o anumită funcționalitate** definită de utilizator.

Tagurile proprii sunt grupate în **librării**.

## Exemple:

- Procesarea unor formulare
- Operații cu baze de date
- Accesarea unor servicii, etc.

# Avantaje

- **Extinderea** ațiunilor standard
- **Reutilizarea** componentelor în aplicații Web diferite
- **Minimizarea** codului Java dintr-o pagină JSP
- **Sintaxa XML**
- **Separarea** activităților de design Web și programare  
**Separation of Concerns (SoC)** = procesul de descompunere a unui sistem în componente cu un grad minim de suprapunere funcțională.

# Caracteristici

- Sunt apelate sub forma:

```
<prefix:numeTag>  
  Continut  
</prefix:numeTag>
```

- Pot fi personalizate prin folosirea atributelor.
- Au acces la obiectele disponibile în pagina JSP.
- Pot crea obiecte ce vor fi vizibile în pagina JSP.
- Mai multe taguri pot comunica între ele.
- Tagurile proprii pot fi imbricate
- Sunt implementate prin *clase* sau *declarativ*



# **Crearea tagurilor proprii folosind clase Java**



# Definirea componentelor

- **Clase Java** care definesc comportamentul tagurilor (*class handler*).
- Un **fișier descriptor** (*tag library descriptor*), care face o mapare între denumirile XML ale tagurilor și clasele ce conțin implementările lor.
- O pagină JSP în care să utilizăm librăria respectivă.

# Crearea clasei *handler*

Definim un tag care va avea ca efect inserarea șirului "Salut !" în fluxul de răspuns al unei pagini JSP.

```
package demo;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;

public class Salut extends SimpleTagSupport {

    public void doTag() throws JspException, IOException {
        JspWriter out = getJspContext().getOut();
        out.print("Salut !");
    }
}
```

# Fișierul descriptor (TLD)



```
<taglib>

  <short-name>
    taguriSimple
  </short-name>

  <description>
    Fișier descriptor pentru biblioteca de taguri
  </description>

  <tag>
    <name>salut</name>
    <tag-class>demo.Salut</tag-class>
    <description> Transmite salutari </description>
    <body-content>empty</body-content>
  </tag>

</taglib>
```



# Folosirea unui tag

- Includerea librăriei de taguri

```
<%@ taglib uri="librarie.tld" prefix="prefix" %>
```

- Referirea tagului

```
<prefix:nume-tag>
```

```
<html>  
<b>Exemplu de folosire a unui tag propriu</b>  
<p>  
<%@ taglib uri="librarie.tld" prefix="simplu" %>  
<simplu:salut/>  
</html>
```

# Identificarea unui tag propriu



## identifier(uri) - location

### web.xml

```
<jsp-config>
  <taglib>
    <taglib-uri>
      http://jakarta.apache.org/tomcat/example-taglib
    </taglib-uri>
    <taglib-location>
      /WEB-INF/jsp/example-taglib.tld
    </taglib-location>
  </taglib>
</jsp-config>
```



# JSP și sintaxa XML

## JSP 1.0 → free-form syntax

```
Hello <%= name %>
```

## JSP 1.2 → XML syntax

```
Hello <jsp:expression>name<jsp:expression>
```

## JSP 2.0 → **.jspx**

```
<?xml version="1.0"?>  
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page">  
  Hello <jsp:expression>name<jsp:expression>  
</jsp:root>
```

# Tipuri de taguri



- **Simple**
- **Cu attribute**
- **Cu corp**
- **Iterative**
- **Imbricate**
- **Care definesc variabile scriptice**
- **Cooperante**



# API

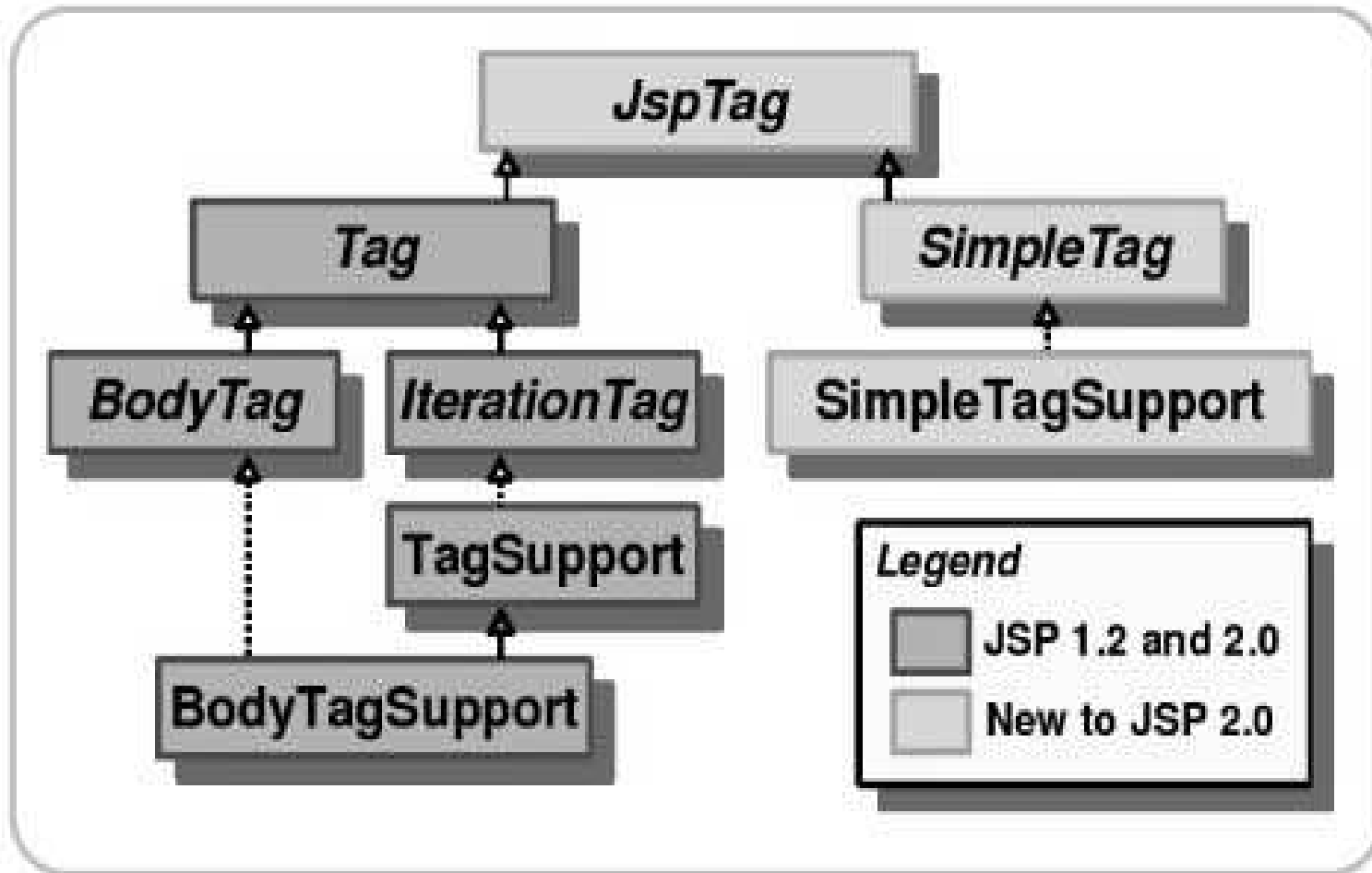


Figure 1 Tag extension class hierarchy

# Taguri simple (SimpleTag)

**Nu au conținut**

Sunt implementate folosind clasa **SimpleTagSupport**.

```
public SimpleTag extends SimpleTagSupport {
    public void doTag() {
        ...
    }
}
```

In fișierul descriptor tagurile simple trebuie să specifice:

```
<body-content>empty</body-content>
```

# Taguri cu attribute statice

Pentru fiecare atribut :

- In clasa corespunzătoare **trebuie să existe metodele de tip getter/setter:**

```
private Tip atribut;  
public Tip getAtribut() { return atribut; }  
public void setAtribut(Tip atribut) { this.atribut = atribut; }
```

- In fișierul descriptor **trebuie să declarăm atributul:**

```
<attribute>  
  <name>atribut</name>  
  <required>true|false|yes|no</required>  
  <rtexprvalue>true|false|yes|no</rtexprvalue>  
  <type>Tip</type>  
</attribute>
```

# Preluarea atributelor

```
...
public class Salut extends SimpleTagSupport {
    private String nume = "";
    public String getNume() {
        return nume;
    }
    public void setNume(String nume) {
        this.nume = nume;
    }
    public void doTag() throws JspException, IOException {
        JspWriter out = getJspContext().getOut();
        out.print("Salut " + nume + " !");
    }
}
<simplu:salut nume="Gigi"/>
```

# Taguri cu attribute dinamice

```
public class DynamicAttributeTag extends SimpleTagSupport
    implements DynamicAttributes {
    private void Map<String, Object> attributes;

    public void setDynamicAttribute(String uri,
        String name, Object value) throws JspException {
        attributes.put(name, value);
    }
    public void doTag()
        throws JspException, IOException {
        // prelucram attributele
    }
}
<tag>
    ...
    <dynamic-attributes>true</dynamic-attributes>
</tag>
```

# Taguri cu corp (SimpleTag)

- Taguri care **nu interacționează** cu conținutul lor
- Taguri care **interacționează** cu conținutul lor
  
- Taguri cu conținut **specific**
- Taguri cu conținut **static** (fără elemente scriptice)

In fișierul descriptor:

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

# Evaluarea conținutului JSP

- Apelăm **getJspBody** pentru a obține conținutul. Acesta este un obiect de tip **JspFragment** și nu poate conține elemente scriptice, fiind format doar din
  - **text static**
  - **acțiuni JSP**
- Apelăm **invoke** pentru a-l evalua.

```
public class Salut extends SimpleTagSupport {
    public void doTag() throws JSPException, IOException {
        getJspBody().invoke(null);
    }
}
```

# Prelucrarea conținutului

Obținem conținutul tagului folosind un flux **StringWriter**:

```
public class Exemplu extends SimpleTagSupport {
    public void doTag() throws JspException, IOException {
        StringWriter sw = new StringWriter();
        getJspBody().invoke(sw);
        getJspContext().getOut().println(sw.toString().toUpperCase());
    }
}
```

# Taguri iterative

Dacă este necesara evaluare iterativă a conținutului atunci:

- Implementăm **SimpleTagSupport** și apelăm metoda **invoke** de câte ori este necesar, sau
- Implementăm **BodyTagSupport** și returnăm **EVAL\_BODY\_AGAIN** din metoda **doAfterBody** de câte ori este necesar.

```
<html>
<b>Exemplu de folosire a unui tag iterativ</b> <br/>
Vor fi generate 10 numere aleatoare. <br/>
<%@ taglib uri="librarie.tld" prefix="simplu" %>
<simplu:repete iteratii="10" >
    <simplu:aleator> <br/>
</simplu:repete>
</html>
```

# Taguri imbricate

```
<prefix:outer-tag>  
  <prefix:inner-tag />  
</prefix:outer-tag>
```

## Exemplu:

```
<table>  
  <tr>  
    <td> <td/>  
  </tr>  
</table>
```

Obținerea părintelui: **findAncestorWithClass**

# Exemplu de imbricare

Definirea unor taguri de tip *if-then-else*:

```
<prefix:if test="<%= expresie %>">  
  <prefix:then>  
    JSP inclus daca expresia este adevarata  
  </prefix:then>  
  
  <prefix:else>  
    JSP inclus daca expresia este falsa  
  </prefix:else>  
</prefix:if>
```

# Legarea claselor imbricate

```
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");
        }
        ...
    }
}
public class IfElseTag extends SimpleTagSupport {
    ...
}
```

# Taguri ce definesc variabile scriptice

Tagurile **pot construi obiecte** care să poata fi folosite în continuare în pagina JSP. Să presupunem ca tagul `salut` va instanția un obiect cu numele `mesaje` din clasa `demo.Mesaje`. In fișierul JSP dorim să utilizam obiectul `mesaje` astfel:

```
<html>
<b>Exemplu de folosire a unei variabile</b>
<%@ taglib uri="librarie.tld" prefix="simplu" %>
<simplu:salut nume="John" limba="english" obiect="mesaje"/>
<p>
<%= mesaje.confirmare() %>
<%= mesaje.reguli() %>
</html>
```

# Declararea variabilelor scriptice

- In clasa tagului, numele obiectului va fi primit ca un atribut: **setObiect**, **getObiect**
- In fișierul descriptor:

```
<tag>
  <variable>
    <name-given>mesaje</name-given>
    <variable-class>demo.Mesaje</variable-class>
    <declare>>true</declare>
    <scope>AT_END|AT_BEGIN|NESTED</scope>
  </variable>
</tag>
```

- In clasa tagului trebuie să creăm obiectul:

```
pageContext.setAttribute(obiect, new demo.Mesaje());
```

# Cooperarea între taguri

Tagurile comunică prin **definirea de obiecte comune**:

- Definite în același context și care pot fi folosite folosind metodele:
  - **getAttribute**
  - **setAttribute**

```
PageContext context = (PageContext) getJspContext();  
context.setAttribute("cheie", valoare);
```

- Definite într-un tag și care sunt comune tuturor tagurilor proprii interne.



# Crearea tagurilor proprii folosind pagini JSP



# Fișiere de taguri

- Fișierele de taguri permit crearea de taguri proprii folosind sintaxa JSP.
- La execuție, fișierul unui tag este transformat într-o clasă de tip *handler* care este compilată automat.
- Extensia recomandată este **.tag**.
- În cadrul unei aplicații Web vor fi plasate:
  - ca fișiere individuale: în `/WEB-INF/tags`
  - ca arhiva jar: `/WEB-INF/lib` (necesită tld)

# Utilizarea unui fișier tag

## **/WEB-INF/tags/salut.tag**

```
<h1> Salut! </h1>
```

## **test.jsp**

```
<%@ taglib tagdir="/WEB-INF/tags" prefix="h" %>
```

```
<html>
```

```
  <h:salut />
```

```
</html>
```

# Directive



Controlează transformarea fișierului `.tag` într-o clasă handler.

- `taglib`
- `include`
- `tag`
- `attribute`
- `variable`



# Declararea tagurilor

```
<%@ tag parametru="valoare" ... %>
```

## Parametri

display\_name

bodycontent="empty | scriptless | tagdependent"

dynamic-attributes

small\_icon, large\_icon

description

import

...

# Declararea atributelor



```
<%@ attribute name="numeAtribut" parametru="valoare" ... %>
```

## Parametri

```
name  
required="true | false | yes | no"  
rtexprvalue="true | false | yes | no"  
description  
type (implicit java.lang.String)  
fragment="true | false | yes | no"
```



# Folosirea atributelor



## /WEB-INF/tags/salut.tag

```
<%@ attribute name="mesaj" required="true" %>
<%@ attribute name="nume" required="true" %>
<h1>
  <%= mesaj %> <%= nume%>!
</h1>
```

## test.jsp

```
<%@ taglib tagdir="/WEB-INF/tags" prefix="h" %>

<html>
  <h:salut mesaj="Bine ai venit" nume="Ioane"/>
  <h:salut mesaj="Hello" nume="folks"/>
</html>
```



# Attribute dinamice



## Declararea

```
<%@ tag dynamic-attributes="attributeMap"%>
```

## Trimiterea atributelor

```
<un:tag>  
  <jsp:attribute name="atributDinamic1">  
    valoareAtribut1  
  </jsp:attribute>  
  <jsp:attribute name="atributDinamic2">  
    valoareAtribut2  
  </jsp:attribute>  
  ...  
</un:tag>
```



# Prelucrarea conținutului



## /WEB-INF/tags/salut.tag

```
<jsp:doBody var="continut" />
<%
String bc = (String) jspContext.getAttribute("continut");
bc = bc.toUpperCase();
%>
<h2>
  <%= bc %> !
</h2>
```

## test.jsp

```
<%@ taglib tagdir="/WEB-INF/tags" prefix="h" %>
<html>
  <h:salut>
    welcome
  </h:salut>
</html>
```

