Java pour le développement d'applications Web : J2EE

JSP 2.0

Mickaël BARON - 2006
mailto:baron.mickael@gmail.com ou mailto:mickael.baron@serli.com

Serli informatique

- Serli : www.serli.com
  - Société de services en informatique
  - Fondé en 1981
  - Située sur le site du Futuroscope, Poitiers
  - Réalisation de logiciels et assistance technique
- Domaines de compétences
  - Systèmes d'informations
  - Embarqué et temps réel
  - Systèmes et réseaux
  - Gestion Electronique de Document (GED, PDM / PLM)
- Plateformes de développement (celles qui concernent les cours)
  - Plateforme Java : Java EE, Java SE, Eclipse
  - Plateforme Microsoft : C#, VB .NET
  - …

Serli informatique

- Equipes impliquées dans l'Open Source
  - Utilisation massive de briques Open Source
  - Formation / diffusion de cours concernant Java et l'Open Source
  - RMLL : 7ème rencontres mondiales du logiciel libre
  - Solutions Linux 2007 : Salon des Solutions Open Source pour l'entreprise
- Membre du consortium ObjectWeb

- Gestion de projets Open Source
  - JaasLounge : interopérabilité JAAS pour Java EE
  - JShaft : gestion de clusters Java EE
  - JWT Gen : tests fonctionnels automatisés

Conception d'un tag personnalisé (2.0)

- Évolutions vers la 2.0 depuis la 1.2
  - Descripteur de balises personnalisées (TLD)
  - Implémentation différente pour le « handler » des balises personnalisées

- Les principales classes des balises personnalisées
  - *SimpleTag* qui est l'interface de base pour écrire un tag
  - *TagExtraInfo* apporte des informations complémentaires sur les tags

Le traitement du corps se fait par la même interface

Peu de modification depuis la version 1.2

- Besoins de conception de deux familles d'élément
  - La classe « handler » qui implémente l'interface *SimpleTag*
  - Le descripteur de la bibliothèque de tag (*.tld)

www.serli.com
serli informatique

Conception d'un tag personnalisé (2.0) : SimpleTag

- L'interface *SimpleTag* permet une implémentation différente de tag JSP réalisée à partir de :
  - *TagSupport*
  - *BodyTagSupport*

La classe *TagExtraInfo* ne change pas. Cependant, nous étudierons les mêmes exemples mais avec *SimpleTag*

- Les classes des balises personnalisées concernées (version 2)
  - *SimpleTag* qui est l'interface de base pour écrire une balise personnalisée
  - *SimpleTagSupport* qui est la classe d'implémentation par défaut de *SimpleTag*
- Présentation avec l'interface *SimpleTag*
  - Reprise du même plan qu'avec *TagSupport* et *BodyTagSupport*
  - Le fichier de description des balises (TLD) évolue légèrement

www.serli.com

serli informatique

Conception d'un tag personnalisé par l'exemple (2.0)

Exemple : « HelloWorld » un classique

```
package monpackage;
...
public class HelloTag extends SimpleTagSupport {
     public void doTag()
               throws JspException, IOException {
          this.getJspContext().getOut().println("Hello World !");
     }
}
```

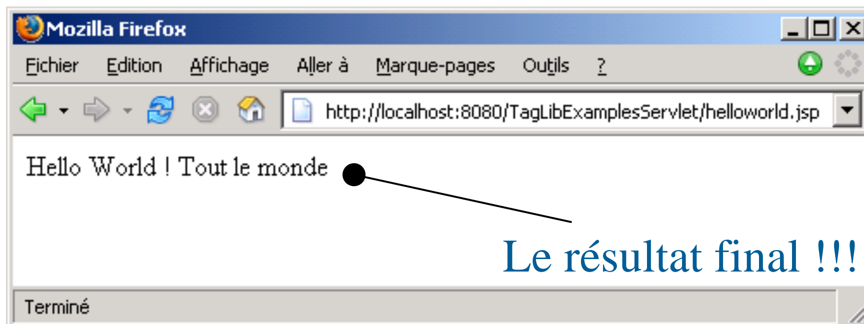
La classe « handler »

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<taglib ...>
     <tlib-version>1.0</tlib-version>
     <jsp-version>2.0</jsp-version>
     <description>
          Bibliothèque de taglibs
     </description>
     <tag>
     <name>hellotag</name>
     <tag-class>monpackage.HelloTag</tag-class>
     <description>
          Tag qui affiche bonjour
     </description>
     <body-content>empty</body-content>
     </tag>
</taglib>
```

Le fichier TLD

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app ...>
<display-name>Permet de gérer des Tags personnalisés</display-name>
<taglib>
     <taglib-uri>monTag</taglib-uri>
     <taglib-location>/WEB-INF/tld/montaglib.tld</taglib-location>
</taglib>
</web-app>
```

Le fichier web.xml



Le résultat final !!!

```
<%@ taglib uri="monTag" prefix="montagamoi" %>

<montagamoi:hellotag /> Tout le monde
```

La JSP avec le nouveau Tag

www.serli.com
serli informatique

Conception d'un tag personnalisé (2.0) : interface Tag

- Chaque balise est associée à une classe qui va contenir les traitements à exécuter lors de leur utilisation
- Pour permettre l'appel à cette classe elle doit obligatoirement implémenter directement ou indirectement l'interface *SimpleTag*
- Préférez l'utilisation de la classe *SimpleTagSupport* qui implémente directement *SimpleTag* (*javax.servlet.jsp.tagext.SimpleTagSupport*)

Une classe « handler » par tag personnalisé et pas un de plus !!!

Avec la version 2, le corps de la balise personnalisée est traité dans une même classe

www.serli.com
serli informatique

Conception d'un tag personnalisé (2.0) : cycle de vie

- L'évaluation d'un tag JSP aboutit aux appels suivants
  - Initialisation de propriétés (*JspContext*, *parent)*
  - Initialisation des attributs s'ils existent
  - Si la balise possède un corps non vide, initialisation de *JspFragment*
  - La méthode *doTag()* est appelée

SimpleTagSupport
+ doTag() **# getJspBody() : JspFragment** **# getJspContext() : JspContext** **+ getParent() : JspTag** **...**

La version 2.0 par rapport à la version 1.2 supprime les méthodes *doStartTag()*, *doInitBody()*, *doAfterBody()*, *doEndTag()*

www.serli.com
serli informatique

Conception d'un tag personnalisé (2.0) : TLD

- Le fichier de description de la bibliothèque de tags décrit une bibliothèque de balises
- Les informations qu'il contient concerne la bibliothèque de tags elle même et concerne aussi chacun des balises qui la compose
- Doit toujours avoir l'extension « .tld »
- Le format des descripteurs de balises personnalisées est défini par un fichier XSD (XML Schema Description)
- En-tête du fichier TLD. Balise ouvrante *<taglib ...>*

```
<?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">
    ...
</taglib>
```

Défini par un fichier XSD

www.serli.com
serli informatique

Conception d'un tag personnalisé (2.0) : TLD

- La première partie du document TLD concerne la bibliothèque
  - *<tlib-version>* : version de la bibliothèque (obligatoire)
  - *<jsp-version>* : version des spécifications JSP (obligatoire)
  - *<short-name>* : nom de la bibliothèque (obligatoire)
  - *<description>* : description de la bibliothèque (optionnelle)
  - *<tag>* : il en faut autant que de balises qui composent la bibliothèque

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<taglib ...>
    <tlib-version>1.0</tlib-version>
    <jsp-version>2.0</jsp-version>
    <description>Bibliothèque de taglibs</description>
    <short-name>TagLibTest</short-name>
    <tag>
        ...
    </tag>
    <tag>
        ...
    </tag>
</taglib>
```

Première balise personnalisée

Seconde balise personnalisée

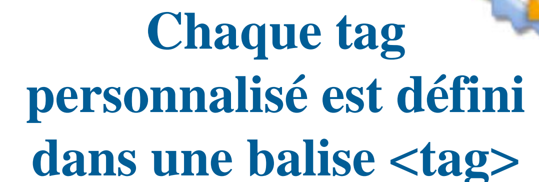
www.serli.com

serli informatique

Conception d’un tag personnalisé (2.0) : TLD

- Chaque balise personnalisée est définie dans la balise <*tag*>
- La balise <*tag*> peut contenir les balises suivantes :
  - <*name*> : nom du tag, il doit être unique dans la bibliothèque (obligatoire)
  - <*tag-class*> : nom de la classe qui contient le handler du tag (obligatoire)
  - <*body-content*> : type du corps du tag (obligatoire)
    - **tagdependent** : l’interprétation du contenu du corps est faite par le tag
    - **empty** : le corps doit obligatoirement être vide
    - **scriptless** : contenir du texte, expressions EL mais pas du code JSP
  - <*attribute*> : décrit les attributs. Autant qu’il y a d’attributs

```
<tag>
    <name>hellotag</name>
    <tag-class>monpackage.HelloTag</tag-class>
    <description>Tag qui affiche bonjour</description>
    <body-content>empty</body-content>
</tag>
</taglib>
```



www.serli.com
serli informatique

Conception d'un tag personnalisé (2.0) : attributs de tag

- Un tag peut contenir des attributs

Tag sans corps avec un attribut appelé « attribut1 »

```
<prefixe:nomDuTag attribut1="valeur" />
```

- La classe « handler » doit définir des modifieurs et des attributs pour chaque attribut du tag

Les attributs ne sont pas obligatoirement de type chaînes de caractères

- Les modifieurs doivent suivre une logique d'écriture de la même manière que pour les Java Beans

```
public class NomDuTag extends SimpleTagSupport {
      private Object attribut1;
      public void setAttribut1(Object p_attribut) {
            this.attribut1 = p_attribut;
      }
      ...
}
```

- Des modifieurs prédéfinis sont utilisés pour initialiser des propriétés du tag (*jspContext, parent et jspBody*)
  - *setJspContext(JspContext)*
  - *setParent(Tag) et setJspBody(JspFragment)*

www.serli.com
serli informatique

Conception d'un tag personnalisé (2.0) : attributs de tag

- Les attributs d'une balise personnalisée doivent être déclarés dans le fichier TLD
- Chaque attribut est défini dans une balise <*attribut*> qui sont contenus dans la balise mère <*tag*>
- La balise <*attribute*> peut contenir les tags suivants :
  - <*name*> : nom de l'attribut utilisé dans les JSP (obligatoire)
  - <*required*> : indique si l'attribut est requis (*true*/*false* ou *yes*/*no*)
  - <*rtexprvalue*> : indique si l'attribut peut-être le résultat d'un tag expression
  - <*type*> : indique le type Java de l'attribut (défaut : *java.lang.String*)

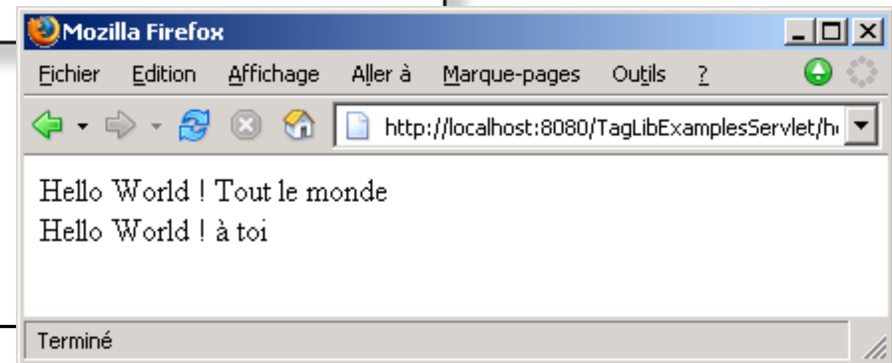
```
<attribute>
    <name>moment</name>
    <required>false</required>
    <rtexprvalue>false</rtexprvalue>
    <type>java.lang.String</type>
</attribute>
```

Conception d'un tag personnalisé (2.0) : attributs de tag

Exemple 1 : « HelloWorld » avec des attributs

```
package monpackage;
...
public class HelloTagAttributs extends SimpleTagSupport {
    private String moment;

    public void setMoment(String p_moment) {
        this.moment = p_moment;
    }

    public int doTag() throws JspException, IOException {
        getJspContext().getOut().println ("Hello World ! " + moment);
    }
}
```

Nouvelle classe pour ce nouveau tag de la même bibliothèque




```
<%@ taglib uri="monTag" prefix="montagamoi" %>

<montagamoi:hellotag/> Tout le monde <br>
<montagamoi:hellotagattributs moment="à toi"/>
```

Ajout d'un attribut au tag

www.serli.com
serli informatique

Conception d'un tag personnalisé (2.0) : attributs de tag

Exemple 1 (suite) : « HelloWorld » avec des attributs

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<taglib ...>
    <tlibversion>1.0</tlibversion>
    <jspversion>2.0</jspversion>
    <info>Bibliothèque de test des taglibs</info>
    <tag>
        <name>hellotag</name>
        <tag-class>monpackage.HelloTag</tag-class>
        <description>Tag qui affiche bonjour</description>
        <body-content>empty</body-content>
    </tag>
    <tag>
        <name>hellotagattributs</name>
        <tag-class>monpackage.HelloTagAttributs</tag-class>
        <description>Bonjour et un attribut</description>
        <attribute>
            <name>moment</name>
            <required>false</required>
            <rtexprvalue>false</rtexprvalue>
        </attribute>
        <body-content>empty</body-content>
    </tag>
</taglib>
```

Deux tags sont définis dans cette bibliothèque

Un seul attribut est défini

www.serli.com
serli informatique

Conception d'un tag personnalisé (2.0) : attributs de tag

- Exemple 1 (suite bis) : omission d'un attribut obligatoire ...

```
<tag>
    ...
    <attribute>
        <name>moment</name>
        <required>true</required>
        <rtexprvalue>false</rtexprvalue>
    </attribute>
</tag>
```

```
<%@ taglib uri="monTag" prefix="montagamoi" %>

<montagamoi:hellotag/> Tout le monde <br>
<montagamoi:hellotagattributs />
```

Utilisation dans une JSP d'un tag avec attribut obligatoire

Apache Tomcat/5.5.4 - Rapport d'erreur - Mozilla Firefox

Fichier Edition Affichage Aller à Marque-pages Outils ?

http://localhost:8080/TagLibExamplesServlet/helloworld.jsp

Etat HTTP 500 -

type Rapport d'exception

message

description Le serveur a rencontré une erreur interne () qui l'a empêché de satisfaire la requête.

exception

```
org.apache.jasper.JasperException: /helloworld.jsp(6,0) D'après le TLD l'attribut moment est obligatoire pour le tag hellotagattributs
        org.apache.jasper.compiler.
        org.apache.jasper.compiler.ErrorDispatcher.dispatch(ErrorDispatcher.java:405)
        org.apache.jasper.compiler.ErrorDispatcher.jspError(ErrorDispatcher.java:234)
        org.apache.jasper.compiler.Validator$ValidateVisitor.visit(Validator.java:693)
        org.apache.jasper.compiler.Node$CustomTag.accept(Node.java:1441)
        org.apache.jasper.compiler.Node$Nodes.visit(Node.java:2163)
        org.apache.jasper.compiler.Node$Visitor.visitBody(Node.java:2213)
        org.apache.jasper.compiler.Node$Visitor.visit(Node.java:2219)
        org.apache.jasper.compiler.Node$Root.accept(Node.java:456)
        org.apache.jasper.compiler.Node$Nodes.visit(Node.java:2163)
        org.apache.jasper.compiler.Validator.validate(Validator.java:1489)
        org.apache.jasper.compiler.Compiler.generateJava(Compiler.java:157)
        org.apache.jasper.compiler.Compiler.compile(Compiler.java:286)
```

Terminé

Une exception se lève quand un attribut obligatoire est omis

www.serli.com
serli informatique

Conception d'un tag personnalisé (2.0) : attributs de tag

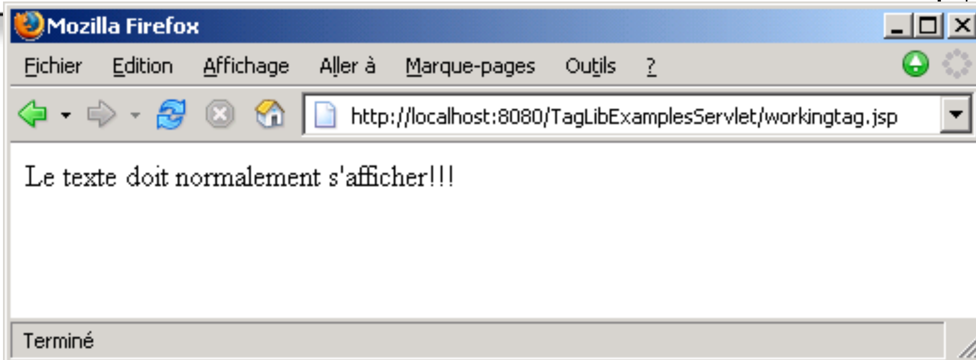
Exemple 2 : évaluation corps et arrêt de l'évaluation de la page

```
public class ExplainWorkingTag extends SimpleTagSupport {
    private String test;
    private String apoca;
    public void setTest(String param) {
        test = param;
    }
    public void setApoca(String param) {
        apoca = param;
    }

    public void doTag() throws JspException, IOException {
        if (test.equals("body")) {
            this.getJspBody().invoke(null);
        } else {
            if (apoca.equals("fin")) {
                // Ne rien faire
            } else {
                throw new SkipPageException();
            }
        }
    }
}
```

Affiche le contenu du corps. Étudié dans la suite …

Peut-être équivalent à EVAL_BODY_INCLUDE de la version 1.2

```
<%@ taglib uri="monTag" prefix="montagamoi" %>

<montagamoi:explainworkingtag test="body" apoca="fin">
Le texte doit normalement s'afficher!!!
</montagamoi:explainworkingtag>

<montagamoi:explainworkingtag test="autre" apoca="fin">
Le texte ne doit pas s'afficher!!!
</montagamoi:explainworkingtag>

<montagamoi:explainworkingtag test="autre" apoca="autre">
Le texte ne doit pas s'afficher!!!
</montagamoi:explainworkingtag>

Le reste de la page ne doit pas être vu.
```



Conception d'un tag personnalisé (2.0) : attributs de tag

Exemple 3 : évaluation de code JSP depuis un attribut

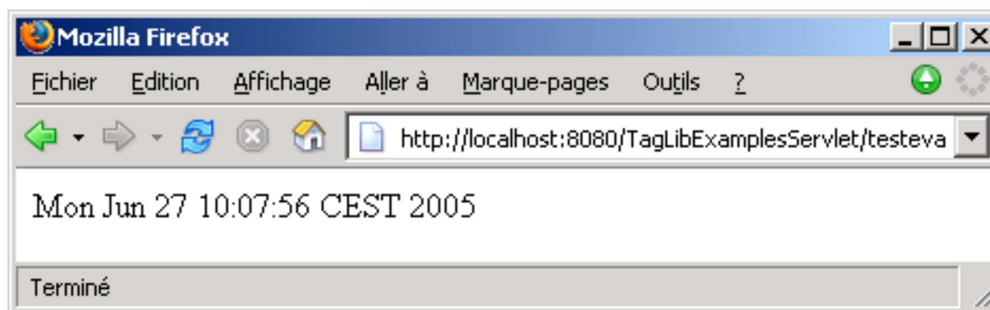
```
<attribute>
    <name>value</name>
    <required>true</required>
    <rtexprvalue>true</rtexprvalue>
</attribute>
```

L'attribut peut recevoir une expression JSP

```
public class EvalExpressionAttribut extends SimpleTagSupport {
    private Object value;
    public void setValue(Object p_value) {
        value = p_value;
    }

    public void doTag() throws JspException, IOException {
        if (value instanceof Date) {
            this.getJspContext().getOut().println((Date)value);
        } else {
            this.getJspContext().getOut().println("N'est pas un objet Date");
        }
    }
}
```

```
<%@ taglib uri="monTag" prefix="montagamoi" %>

<montagamoi:evalexpressattribut value="<%= new java.util.Date() %>" />
```

Un objet autre que *String* peut-être envoyé



Conception d'un tag personnalisé (2.0) : variables implicites

- Les balises personnalisées ont accès aux variables implicites de la JSP dans laquelle ils s'exécutent via un objet de type Jsp*Context*
- Utilisation de la méthode *JspContext getJspContext()*
- La classe *JspContext* définit plusieurs méthodes
  - *JspWriter getOut()* : accès à la variable *out* de la JSP
  - *Object getAttribute(String)* : retourne objet associé au paramètre (scope à *page*)
  - *Object getAttribute(String, int)* : retourne objet avec un scope précis
  - *setAttribute(String, Object)* : associe un nom à un objet (scope à *page*)
  - *setAttribute(String, Object, int)* : associe un nom à un objet avec un scope
  - *Object findAttribute(String)* : cherche l'attribut dans les différents scopes
  - *removeAttribute(String)* : supprime un attribut, ...

www.serli.com
serli informatique

Conception d'un tag personnalisé (2.0) : variables implicites

- Les valeurs du scope est défini dans *PageContext*
  - PAGE_SCOPE : attribut dans le scope *page*
  - REQUEST_SCOPE : attribut dans le scope *request*
  - SESSION_SCOPE : attribut dans le scope *session*
  - APPLICATION_SCOPE : attribut dans le scope *application*
- Exemples d'utilisation des méthodes de *PageContext*

```
getJspContext().setAttribute("toto", new Date(), PageContext.PAGE_SCOPE);
```

Création d 'un attribut « toto » avec la valeur d'une *Date* dans le scope « page »

```
getJspContext().findAttribute("toto");
```

Cette méthode cherche l'attribut « toto » dans tous les scopes en commençant par *page*, *request*, *session* et *application*

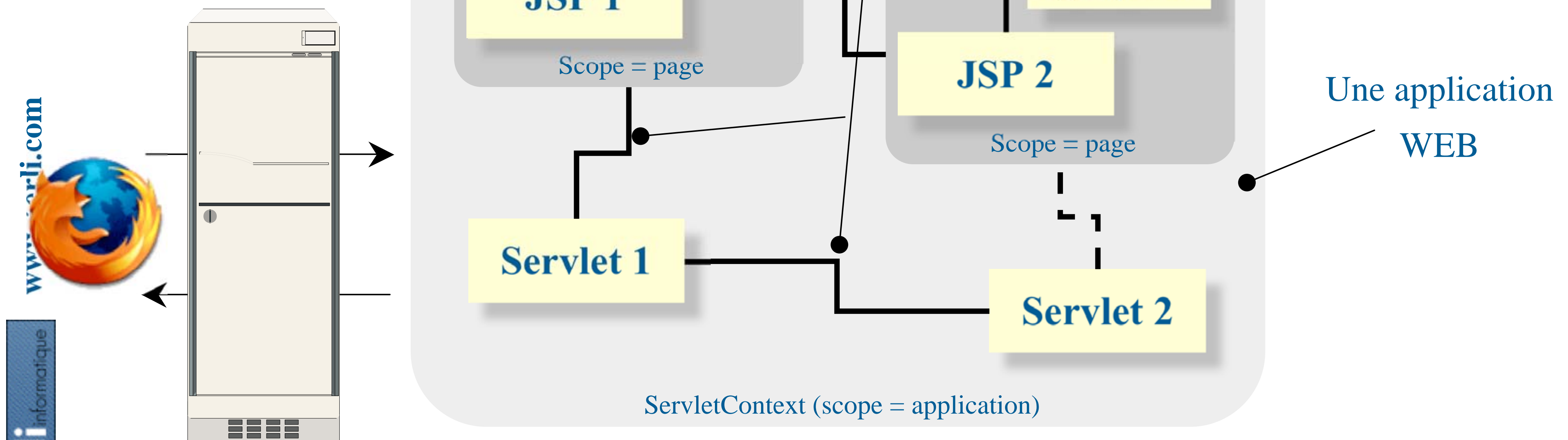
```
getJspContext().getAttribute("toto", PageContext.PAGE_SCOPE);
```

Récupère l'attribut « toto » dans le scope « page »

www.serli.com
serli informatique

Conception d'un tag personnalisé (2.0) : variables implicites

- Possibilité de récupérer la valeur d'un attribut selon son scope et ainsi de communiquer entre une JSP un tag et une Servlet



Conception d'un tag personnalisé (2.0) : variables implicites

- Exemple 1 : communication entre Bean et balise personnalisée

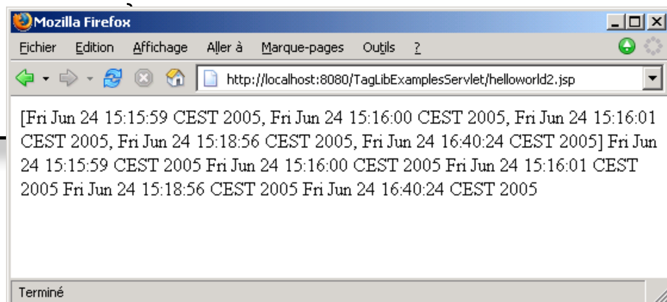
```
<jsp:useBean id="mon_bean" class="java.util.ArrayList" scope="application" />
<%@ taglib uri="monTag" prefix="montagamoi" %>

<% mon_bean.add(new java.util.Date()); %>

<montagamoi:hellotagarraylist name="mon_bean" />
```

Définition d'un Java Bean dans le contexte de l'application WEB

```
public class HelloTagArrayList extends SimpleTagSupport {
    private String mon_bean;
    public void setName(String p_bean) {
        this.mon_bean = p_bean;
    }

    public void doTag() throws JspException, IOException {
        Object my_object = findAttribute(mon_bean);
        if (my_object != null ) {
            ArrayList my_array_list = (ArrayList)my_object;
            for (int i = 0; i < my_array_list.size(); i++) {
                getJspContext().getOut().println(my_array_list.get(i));
            }
        } else {
            getJspContext().getOut().println("Y a rien");
        }
    }
}
```

L'attribut « name » permet d'indiquer l'identifiant du Bean

Sachant que l'instance du Java Bean est défini dans le scope application



www.serli.com
serli informatique

Conception d'un tag personnalisé (2.0) : variables implicites

- Exemple 1 (suite) : plusieurs solutions pour y arriver ...

```
<jsp:useBean id="mon_bean" class="java.util.ArrayList" scope="application" />
<%@ taglib uri="monTag" prefix="montagamoi" %>

<% mon_bean.add(new java.util.Date()); %>

<montagamoi:hellotagarraylist name=<%= mon_bean %> />
```

Évaluation d'expression JSP

```
public class HelloTagArrayList2 extends TagSupport {
    private Object bean;
    public void setBean(Object my_bean) {
        this.bean = my_bean;
    }
    public void doTag() throws JspException, IOException {
        if (bean instanceof ArrayList) {
            if (bean != null ) {
                ArrayList my_array_list = (ArrayList)bean;
                for (int i = 0; i < my_array_list.size(); i++) {
                    getJspContext().getOut().println(my_array_list.get(i));
                }
            } else {
                getJspContext().getOut().println("Y a rien");
            }
        }
    }
}
```

Il faut s'assurer que l'objet envoyé en attribut est du type *ArrayList*

Préférez cette solution à la première, elle est moins dépendante que la première version

www.serli.com
serli informatique

Conception d'un tag personnalisé (2.0) : variables implicites

- Exemple 2 : collaboration de tag personnalisées « switch …case »

```
<%@ taglib uri="monTag" prefix="montagamoi" %>

<montagamoi:switchtag test="3">
    <montagamoi:casetag value="0">Zéro</montagamoi:casetag>
    <montagamoi:casetag value="1">Un</montagamoi:casetag>
    <montagamoi:casetag value="2">Deux</montagamoi:casetag>
    <montagamoi:casetag value="3">Trois</montagamoi:casetag>
</montagamoi:switchtag>
```

Simulation de « switch case » par l'intermédiaire de balises

```
public class SwitchTag extends SimpleTagSupport {
    private String test;
    public void doTag() throws JspException, IOException {
        getJspBody().invoke(null);
    }

    public void setTest(String p_value) {
        test = p_value;
    }

    public boolean isValid(String caseValue) {
        if (test == null) return false;
        return(test.equals(caseValue));
    }
}
```

Le corps du tag est évalué

Initialise l'attribut « test »

Vérifie que « test » est le même que celui du tag enfant

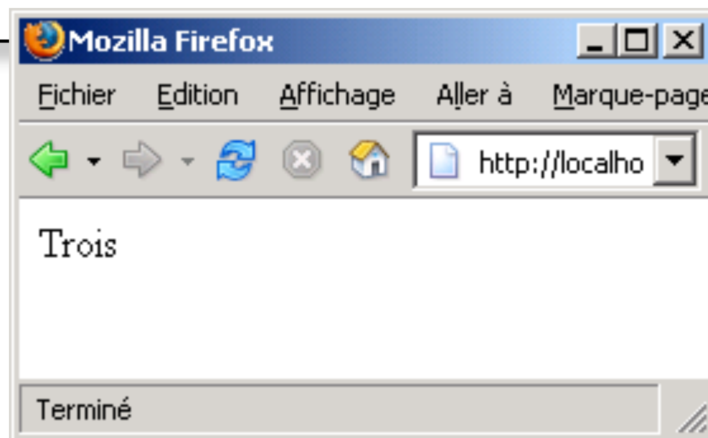
Conception d'un tag personnalisé (2.0) : variables implicites

Exemple 2 (suite) : collaboration de balises personnalisées ...

```
public class CaseTag extends TagSupport {
    public String value;

    public void setValue(String p_value) {
        this.value = p_value;
    }

    public void doTag() throws JspException, IOException {
        if (getParent() instanceof SwitchTag) {
            SwitchTag parent = (SwitchTag)getParent();
            if (parent.isValid(this.value)) {
                this.getJspBody().invoke(null);
            }
        } else {
            throw new JspException("Case doit être à l'intérieur du tag Switch");
        }
    }
}
```

Vérifie que « test » du tag parent est le même que « value » du tag enfant

Affiche ou non le corps de la balise enfant

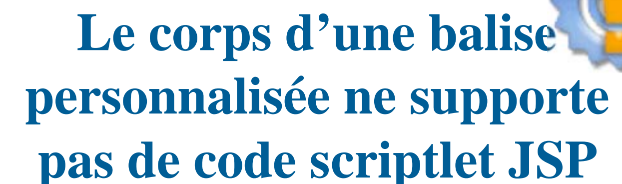


www.serli.com

serli informatique

Conception d'un tag personnalisé (2.0) : corps du Tag

- La méthode *doTag()* traite son corps via l'objet *JspFragment* renseignée par le serveur d'application
- Utilisation de la méthode *JspFragment getJspBody()* permettant de récupérer l'objet *JspFragment*
- L'objet *JspFragment* peut être évalué autant de fois que nécessaire grâce à la méthode *invoke(Writer)* qui écrit le résultat dans le *Writer* spécifié
- Un *Writer* peut être de différents types
  - *StringWriter* : flux de chaînes de caractères
  - *OutputStreamWriter*
  - *PrintWriter*
  - ...



www.serli.com
serli informatique

Conception d'un tag personnalisé (2.0) : corps du Tag

- A la différence de la version 1.2, l'implémentation par *SimpleTag* permet de bufériser le traitement du corps dans un seul appel de la méthode *doTag()*

- Pour retourner uniquement le contenu du corps il existe plusieurs façon
  - Utilisation d'un *Writer* « null »

Cette écriture a déjà été aperçue dans les exemples précédents

```
getJspBody().invoke(null)
```

  - Utilisation d'un *Writer* de type *StringWriter*

Création d'un *Writer* de chaînes de caractères

Place le contenu du corps dans le *Writer*

```
StringWriter mon_buffer = new StringWriter();
getJspBody().invoke(mon_buffer);
getJspContext().getOut().println(mon_buffer);
```

Affiche le contenu du *Writer* dans la réponse

www.serli.com
serli informatique

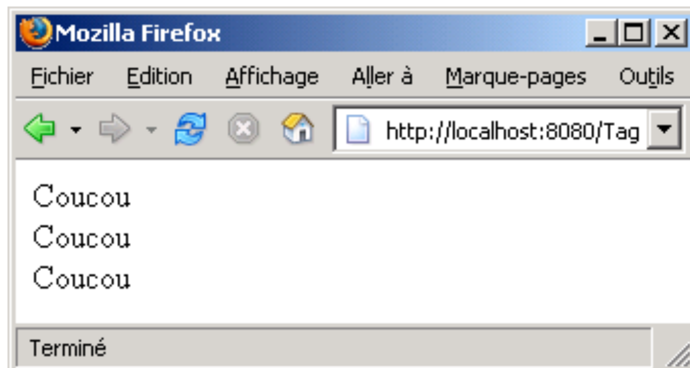
Conception d'un tag personnalisé (2.0) : itération sur le corps

- Exemple : itération sur le corps du tag ...

```
public class IterateSimpleTag extends BodyTagSupport {
        private int count = 0;

    public void setCount(int i) {
        count = i;
    }

    public void doTag() throws JspException, IOException {
        for (int i = 0; i < count; i++) {
            getJspBody().invoke(null);
        }
    }
}
```

A chaque itération le contenu du corps est écrit dans la page JSP

```
<%@ taglib uri="monTag" prefix="montagamoi" %>

<montagamoi:iteratesimpletag count="3">
    Coucou<br>
</montagamoi:iteratesimpletag>
```



Écriture plus simple que l'implémentation fournie par *BodyTagSupport*

www.serli.com
serli informatique

Conception d'un tag personnalisé (2.0) : itération sur le corps

- Exemple (bis) : itération sur le corps du tag avec modification

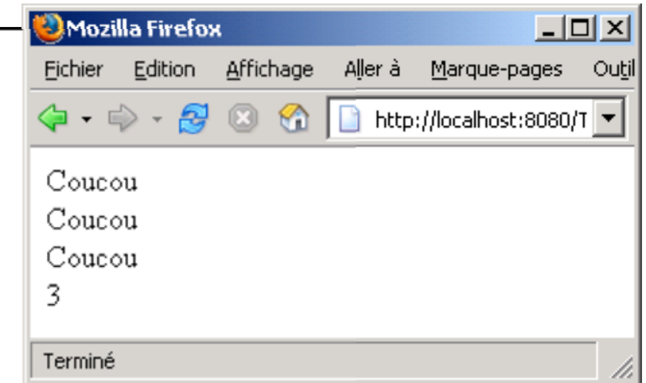
```
public class IterateSimpleTag extends BodyTagSupport {
    private int count = 0;

    public void setCount(int i) {
        if (i <= 0) {
            i = 1;
        }
        count = i;
    }

    public void doTag() throws JspException, IOException {
        StringWriter mon_writer = new StringWriter();
        this.getJspBody().invoke(mon_writer);
        String contenu = mon_writer.toString();

        for (int i = 0; i < count - 1 ; i++) {
            mon_writer.append(contenu);
        }
        mon_writer.append(String.valueOf(count));
        getJspContext().getOut().println(mon_writer);
    }
}
```

Utilisation d'un Writer de type *StringWriter*

Récupère le contenu du corps

Ajoute le contenu du corps « count - 1 » fois

Affiche le résultat bufférisé sur la page JSP




```
<%@ taglib uri="monTag" prefix="montagamoi" %>

<montagamoi:iteratetag count="3">
    Coucou<br>
</montagamoi:iteratetag>
```

www.serli.com
serli informatique

Conception d’un tag personnalisé (2.0) : itération sur le corps

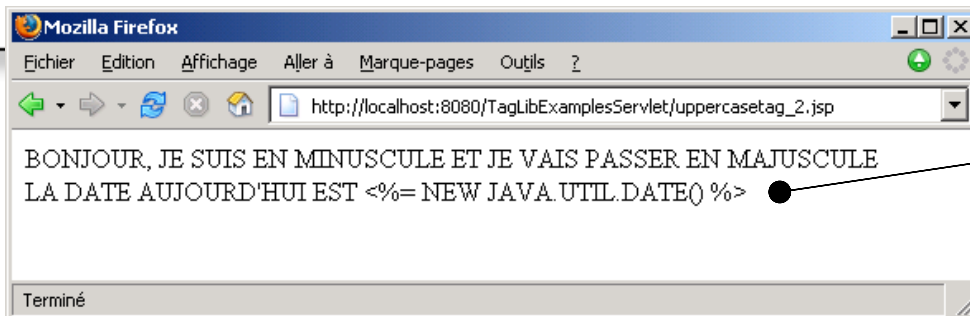
- Exemple (bis) : itération sur le corps du tag avec modification

```java
public class UpperCaseTag extends SimpleTagSupport {
    public void doTag() throws JspException, IOException {
        StringWriter mon_writer = new StringWriter();
        getJspBody().invoke(mon_writer);
        String ma_chaine = mon_writer.toString();

        getJspContext().getOut().println(ma_chaine.toUpperCase());
    }
}
```

Le contenu du *Writer* est récupéré puis modifié en majuscule

```jsp
<%@ taglib uri="monTag" prefix="montagamoi" %>

<montagamoi:uppercasetag>
Bonjour, je suis en minuscule et je vais passer en majuscule <br>
La date aujourd'hui est <%= new java.util.Date() %>
</montagamoi:uppercasetag>
```

```xml
<tag>
<name>uppercasetag</name>
<tag-class>monpackage.UpperCaseTag</tag-class>
<description>Tag qui effectue des modifs sur le corps d'un Tag</description>
<body-content>tagdependent</body-content>
</tag>
```



Non évalués puisque le code JSP n’est plus interprétable dans le corps (2.0)
Solution : EL

www.serli.com
serli informatique

Expressions Languages (2.0) : justifications ...

- L'utilisation de l'interface *SimpleTag* de la version 2.0 des JSP ne permet plus d'exploiter du code JSP dans le corps des balises personnalisées
- Les raisons :
  - Limiter la présence du langage Java (non spécialiste)
  - Meilleure lisibilité, le code se limite au nom des beans et de ses propriétés

```
<%= page.getAttribute("personne").getNom() %>
```

```
${page["personne"].nom}
```

Version (Java) avec tag expression et version (EL) de type « script »

- Les **Expressions Languages** (EL) permettent de manipuler les données au sein d'une page JSP (essentiellement les Beans)
- Une EL permet d'accéder simplement aux beans des différents scopes de l'application (*page*, *request*, *session* et *application*)

www.serli.com
serli informatique

Expressions Languages (2.0) : Qu'est ce que c'est ...

- Forme d'une expression EL

```
${ expression}
```

- Une expression correspond à l'expression à interpréter. Elle peut-être composée de plusieurs termes séparés par des opérateurs

```
${terme1 opérateur terme2}
${opérateur-unaire terme}
${terme1 opérateur terme2 opérateur terme3 ...}
```

- Termes peuvent être :
  - un type primaire
  - un objet implicite
  - un attribut d'un scope de l'application web
  - une fonction EL

Non étudiée dans la suite ...

www.serli.com
serli informatique

Expressions Languages (2.0) : comment les utiliser ?

- Par défaut les expressions EL sont ignorées s'il n'y a pas de fichier web.xml. Utilisation de la directive *page* pour les activer (false)

```
<%@ page isELIgnored="false" %>
```

- Les expressions EL peuvent être utilisées dans
  - les attributs des tags personnalisés (existants ou les vôtre)
  - le corps des tags personnalisés
  - dans la page JSP (hors tags)

```
${lebeanalui}
<prefix:montagamoi param1="toto" param2="${lebeanalui}" >
    ${lebeanalui}
</prefix:montagamoi>
```

- Les types primaires de Java peuvent être utilisés dans les EL
  - *null* : la valeur null
  - *java.lang.Long* : 17
  - *java.lang.String* : « ma chaîne à moi »
  - *java.lang.Boolean* : true ou false, ...

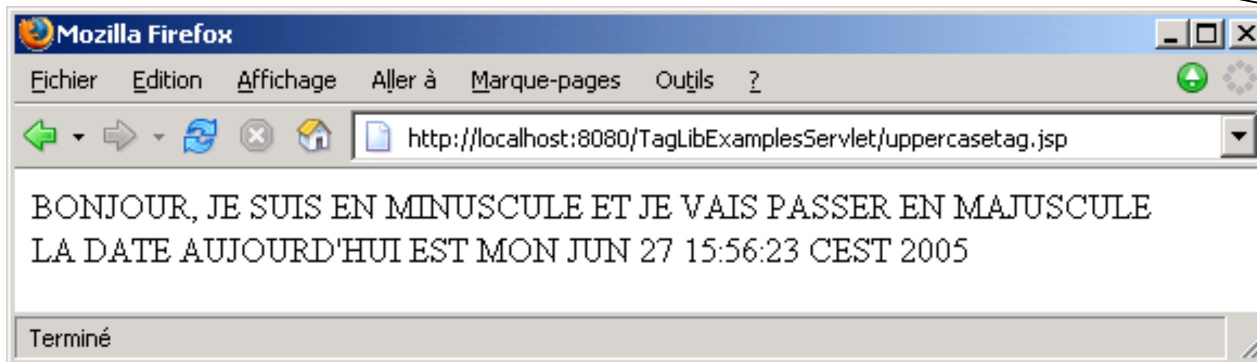
www.serli.com
serli informatique

Expressions Languages (2.0) : objet implicites

➤ Exemple : EL et balise personnalisée ...

```
<%@ page isELIgnored="false" %>
<%@ taglib uri="monTag2" prefix="montagamoi" %>
<jsp:useBean id="mon_bean" class="java.util.Date" scope="page" />

<montagamoi:uppercasetag>
Bonjour, je suis en minuscule et je vais passer en majuscule <br>
La date aujourd'hui est ${pageScope["mon_bean"]}
</montagamoi:uppercasetag>
```

```
<tag>
<name>uppercasetag</name>
<tag-class>monpackage.UpperCaseTag</tag-class>
<description>Tag qui effectue des modifs sur le corps d'un Tag</description>
<body-content>scriptless</body-content>
</tag>
```



Le *<body-content>* peut maintenant être à *scriptless* puisque le corps contient une EL ...

www.serli.com
serli informatique

Expressions Languages (2.0) : objet implicites

- Un certains nombres d'objets implicites permettent d'accéder aux différents composants d'une page JSP
  - *pageContext* : accès à l'objet *PageContext* (*request*, *response*, …)
  - *pageScope["…"]* : accès aux différents attributs du scope « page »
  - *requestScope["…"]* : accès aux différents attributs du scope « request »
  - *sessionScope["…"]* : accès aux différents attributs du scope « session »
  - *applicationScope["…"]* : accès aux différents attributs du scope « application »
  - *param["…"]* : accès aux paramètres de la requête HTTP
  - *paramValues* : paramètres de la requête sous la forme d'un tableau *String*
  - *header["…"]* : accès aux valeurs d'un en-tête HTTP
  - *headerValues* : accès aux en-têtes de la requête sous forme d'un tableau *String*
  - *cookie["…"]* : accès aux différents cookies, …
- Les objets avec crochet sont tous de type *Map*

www.serli.com
serli informatique

Expressions Languages (2.0) : objet implicites

- Lors de l'évaluation d'un terme, si celui-ci n'est ni un type primaire, ni un objet implicite, le conteneur JSP recherchera alors un attribut du même nom dans les différents scopes (*page*, *request*, …)

```
${ nom}
```

Si « nom » a été défini dans un scope ces deux expressions font la même chose

```
<%= pageContext.findAttribute("name") %>
```

L'instruction EL est plus simple est moins langage

```
${ sessionScope[ "nom"] }
```

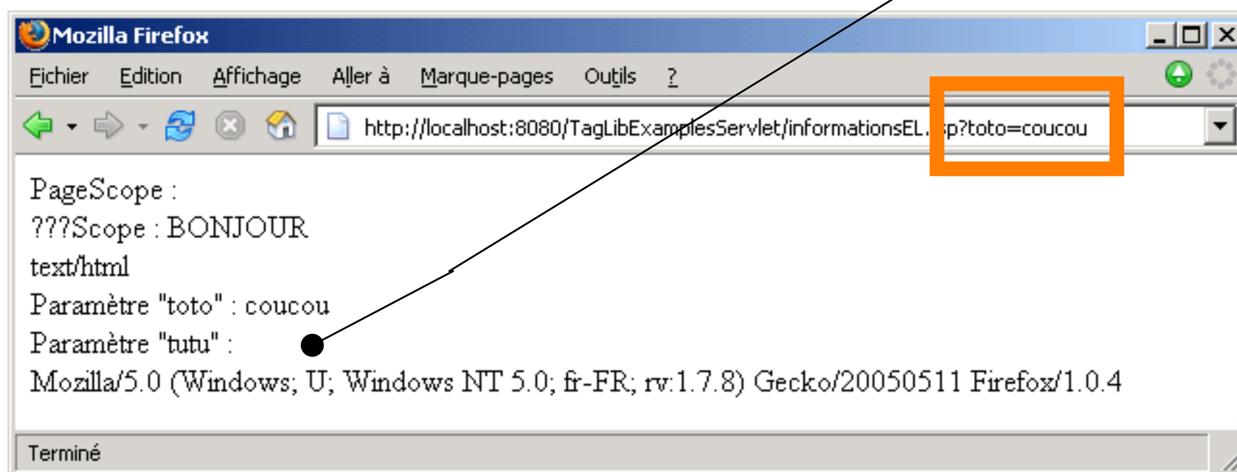
Cette seconde instruction recherche uniquement l'attribut « nom » dans le scope « session ». S'il n'existe pas retourne chaîne vide

Expressions Languages (2.0) : objet implicites

Exemple : lecture d'informations ...

```
<%@ page isELIgnored="false" %>

<% pageContext.setAttribute("mot1", "BONJOUR", PageContext.SESSION_SCOPE); %>
<% pageContext.setAttribute("mot2", "AuRevoir", PageContext.APPLICATION_SCOPE); %>

PageScope : ${pageScope["mot1"]} <br>
???Scope : ${mot2} <br>

${pageContext.response.contentType} <br>
Paramètre "toto" : ${param["toto"]} <br>
Paramètre "tutu" : ${param["tutu"]} <br>
${header["user-agent"]} <br>
```

Si le paramètre n'existe pas, pas de renvoie d'objet « null » mais d'une chaîne de caractères vide



Les expressions EL permettent de gérer plus facilement les valeurs « null »

www.serli.com
serli informatique

Expressions Languages (2.0) : accès aux propriétés des objets

- Il existe trois catégories d'objets définies par les expressions EL
  - les objets standards (tout autres objets)
  - les objets indexées (tableau Java, ou objet de type *List*)
  - les objets mappés (objet de type *Map*)
- Les objets sont construits suivant le modèle des Beans
- L'accès se fait par réflexivité, c'est-à-dire qu'il n'y a pas besoin de connaître le type de l'objet pour accéder à ses propriétés
- Selon le type d'objet, les règles d'accès à ces propriétés divergent malgré une syntaxe similaire

Accès aux propriétés des objets (2.0) : objets standards

- Utilisation d'un accesseur spécifique portant le nom de la propriété
  - L'accès à la propriété « name » se fait par l'accesseur *getName()*. Cette méthode est recherchée puis appelée
- Pour accéder à la propriété deux approches :
  - le point : `${ objet.name }`
  - le crochet : `${ objet[ "name"] }` ou `${ objet['name'] }`

Préférez l'usage du point afin d'éviter les confusions avec les autres types d'objets

```
<%
    MonBeanAMoi mon_bean = (MonBeanAMoi)pageContext.findAttribute("name");
    if (mon_bean != null)
        out.println(bean.getName());
%>
```

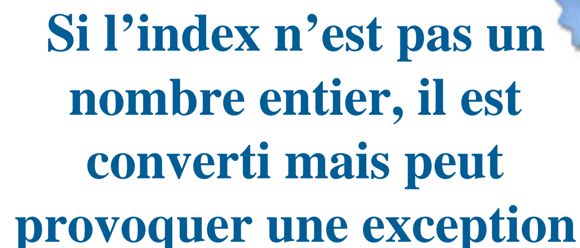
Le code qui correspond à cette écriture

S'il n'existe pas d'accesseur pour la propriété, une exception sera lancée

Accès aux propriétés des objets (2.0) : objets indexées

- Accéder à une propriété indexée d’un tableau ou une liste
- Utilisation d’un accesseur spécifique portant en paramètre l’indice nom de la propriété (méthode *get(int)* de l’interface *List*)
- Pour accéder à la propriété une approche par crochet :

```
${  objet[ 0]  }
${  objet[ 1]  }
${  objet[ 5]  }
${  objet[ "2"]  }
${  objet[ "4"]  }
${  objet[ "5"]  }
```



www.serli.com
serli informatique

Accès aux propriétés des objets (2.0) : objets mappés

- Accéder à une propriété d'un objet mappé. L'accès se fait par une étiquette

- Utilisation d'un accesseur spécifique portant en paramètre une clé permettant d'accéder à la propriété (méthode *get(Object)* de l'interface *Map*)

- Pour accéder à la propriété utilisation de l'approche par crochet :

```
${ objet["clef1"] }
${ objet["clef2"] }
${ objet["clef4"] }
```

www.serli.com
serli informatique

Expressions Languages (2.0) : opérateurs

- Il est également possible d'utiliser des opérateurs dans une expression EL
- Il s'agit des mêmes opérateurs que ceux du langage Java, mis à part que certains possèdent un équivalent textuel afin d'éviter des conflits
- Les opérateurs s'opèrent sur deux termes et prennent la forme suivante :

```
Terme1 operateur Terme2
```

- Opérateurs arithmétiques : « + », « - », « * », « / », …
- Opérateurs relationnels : « == », « != », « < », « > », « <= »
- Opérateurs logiques : « && », « || », « ! »
- Autres : « ? : », « test »

www.serli.com
serli informatique

Conception d'un tag personnalisé (2.0) : TagExtraInfo

- La classe *TagExtraInfo* permet de fournir des informations supplémentaires sur la balise au moment de la **compilation** de la JSP

- Package et classe *javax.servlet.jsp.tagext.TagExtraInfo*

- Elle définit principalement trois méthodes :

  - *TagInfo getTagInfo()* : accéder aux informations sur le tag contenu dans le descripteur de taglib (TLD)

  - *VariableInfo[] getVariableInfo(TagData)* : permet de mapper des éléments de scopes vers des variables de script dans la page JSP

  - *boolean isValid(TagData)* : permet de valider la balise avant même que la classe de la balise (« handler ») soit exécutée

www.serli.com
serli informatique

Conception d'un tag personnalisé (2.0) : variables de script

- Exemple : création de variables de script (sans *TagExtraInfo*)

```
public class VariableScript extends TagSupport {
    private String name = null;

    public void setName(String p_string) {
        this.name = p_string;
    }

   public void doTag() throws JspException, IOException {
       getJspContext().setAttribute(name, new Date());
       getJspBody().invoke(null);
    }
}
```

Définition d'un attribut par l'intermédiaire de l'objet implicite *jspContext* (scope = page)

Utilisation d'une expression EL pour accéder à l'attribut (scope = page)

```
<%@ taglib uri="monTag" prefix="montagamoi" %>

<montagamoi:variablescript name="value" >
   ${value}
</montagamoi:variablescript>
${value} <br>
<%= pageContext.getAttribute("value") %>
```

Donne le même résultat pour une EL et un tag expression

www.serli.com
serli informatique

Conception d'un tag personnalisé (2.0) : variables de script

- Exemple (bis) : création de variables de script (avec *TagExtraInfo*)

```
public class VariableScript extends TagSupport {
    private String name = null;

    public void setName(String p_string) {
        this.name = p_string;
    }

    public void doTag() throws JspException, IOException {
        getJspContext().setAttribute(name, new Date());
        getJspBody().invoke(null);
    }
}
```

Définition d'un attribut par l'intermédiaire de l'objet implicite *jspContext* (scope = page)

```
<%@ taglib uri="monTag" prefix="montagamoi" %>

<montagamoi:variablescript name="value" >
  ${value}
</montagamoi:variablescript>
${value} <br>
<%= value %>
```

Utilisation d'une expression EL pour accéder à l'attribut (scope = page)

« value » est ainsi défini comme une variable de script

A suivre ...

Conception d'un tag personnalisé (2.0) : TagExtraInfo

- La méthode *getVariableInfo(TagData)* s'occupe de mapper les éléments des attributs vers des variables de script présent dans la JSP
- Retourne un objet de type *VariableInfo* qui doit contenir
  - le nom de la variable de script
  - le nom du type de la variable
  - un booléen qui indique si la variable doit être déclarée (vraie) ou si on doit réutiliser une variable déjà déclarée (faux)
  - La zone de portée de la variable
    - *int AT_BEGIN* : de la balise ouvrante à la fin de la JSP
    - *int AT_END* : de la balise fermante à la fin de la JSP
    - *int NESTED* : entre les balises ouvrantes et fermantes

Conception d'un tag personnalisé (2.0) : TagExtraInfo

- Un objet *TagInfo* est utilisé pour accéder aux informations du descripteur de taglib (TLD)
- Il définit plusieurs méthodes :
  - *String getTagName()* : nom de la balise personnalisée
  - *TagAttributeInfo[] getAttributes()* : information sur les attributs
  - *String getInfoString()* : information concernant la balise personnalisée

```
...
TagAttributeInfo[] tab_attribute = this.getTagInfo().getAttributes();
for (int i = 0; i < tab_attribute.length; i++) {
    System.out.println(tab_attribute[i].getName());
}
...
```

Récupère par l'intermédiaire du *TagInfo* la liste de tous les attributs

Affiche l'intégralité des noms des attributs d'un tag

www.serli.com

serli informatique

Conception d'un tag personnalisé (2.0) : TagExtraInfo

- Un objet *TagData* est utilisé pour accéder aux valeurs des attributs d'une balise personnalisée

- Rappel : c'est un objet paramètre qui se trouve dans les méthodes
  - *VariableInfo[] getVariableInfo(TagData)*
  - *boolean isValid(TagData)*

- Définit plusieurs méthodes :
  - *Object getAttribute(String)* : la valeur d'un attribut
  - *setAttribute(String, Object)* : modifie la valeur d'un attribut

www.serli.com
serli informatique

Conception d'un tag personnalisé (2.0) : variables de script

Exemple (bis) : création de variables de script (avec *TagExtraInfo*)

```
<%@ taglib uri="monTag" prefix="montagamoi" %>
<montagamoi:variablescript name="value" >
${value} <br>
</montagamoi:variablescript>
${value} <br>
<%= value %>
```

Définition d'un attribut dans le scope « page »

```
public class VariableScriptInfo extends TagExtraInfo {
    public VariableInfo[] getVariableInfo(TagData arg0) {
        VariableInfo[ ] vi = new VariableInfo[ 1] ;

        TagAttributeInfo[ ] tab_attribute = this.getTagInfo().getAttributes();
        vi[ 0] = new VariableInfo(
            (String)arg0.getAttribute(tab_attribute[0].getName()),
            "java.util.Date",
            true,
            VariableInfo.AT_BEGIN);
        return vi;
    }
}
```

Récupère l'intégralité des attributs du tag

« name »

« value »

Type la variable de script

Déclaration de la variable de script

La variable a une portée complète du début jusqu'à la fin de la page JSP

www.serli.com
serli informatique

Conception d'un tag personnalisé (2.0) : TagExtraInfo

- Il faut déclarer la classe de type *TagExtraInfo* dans le descripteur de balise personnalisée

- Elle se fait par l'intermédiaire de la balise <*teiclass*>

```
<tag>
    <teiclass>package.ClasseTagExtraInfo</teiclass>
    ...
</tag>
```

- Pour finir l'exemple de la création de variables de script

```
<tag>
<name>variablescript</name>
<tag-class>monpackage.VariableScript</tag-class>
<teiclass>monpackage.VariableScriptInfo</teiclass>
<description>Tag qui montre la déclaration d'une variable de script</description>
<attribute>
<name>name</name>
<required>true</required>
</attribute>
<body-content>scriptless</body-content>
</tag>
```

Impossibilité d'utiliser du code JSP dans le corps d'un tag avec l'interface *SimpleTag*

www.serli.com
serli informatique

Conception d'un tag personnalisé (2.0) : TagExtraInfo

- Possibilité de valider dynamiquement les attributs de la balise avant qu'il ne soit exécuté

- Utilisation de la méthode *isValid()* qui est appelée à la compilation de la page JSP

- Elle ne permet pas de vérifier la valeur des attributs dont la valeur est le résultat d'un tag expression *<%= object %>* ou d'une scriplet

- Deux intérêts :
  - Validation effectuée pour tous les tags à la compilation
  - Vérification peut être longue mais est faite uniquement à la compilation

www.serli.com
serli informatique

Conception d'un tag personnalisé (2.0) : TagExtraInfo

- Exemple : vérification des attributs

```
<%@ taglib uri="monTag" prefix="montagamoi" %>
<montagamoi:variablescript name="value" >
    ...
</montagamoi:variablescript>
```

L'attribut « name » contient une chaîne de caractères

```
public class VariableScriptInfo extends TagExtraInfo {
    public boolean isValid(TagData arg0) {
        if (arg0.getAttributeString("name").equals("")) {
            System.out.println("Problème dans le tag name");
            return false;
        }
        return true;
    }
```

Affichage dans la console avant l'exécution de la page JSP

```
<%@ taglib uri="monTag" prefix="montagamoi" %>
<montagamoi:variablescript name="<%= 'coucou' %>" >
    ...
</montagamoi:variablescript>
```

Impossibilité de vérifier le contenu d'un tag expression avec *TagExtraInfo*

www.serli.com

serli informatique

Conception d'un tag personnalisé (2.0) : attributs dynamiques

- La version 2.0 vient combler un des principaux défauts des versions précédentes : la gestion d'attributs dynamiques
- Il est déjà possible de rendre un attribut facultatif ou pas mais les attributs doivent obligatoirement
  - posséder une méthode modifieur « *setAttribut(…)* »
  - être déclarer dans le fichier TLD
- L'interface *DynamicAttributes* résout ce problème avec la méthode

```
void setDynamicAttribute(String uri, String localname, Object value)
```

Espace de nommage de l'attribut — Nom de l'attribut — Valeur de l'attribut

Les attributs dynamiques peuvent être utilisés conjointement avec des attributs classiques

Conception d'un tag personnalisé (2.0) : attributs dynamiques

- Pour pouvoir utiliser les attributs dynamiques, il faut donc que la classe « handler » implémente l'interface *DynamicAttributes* en plus d'une des interfaces *JspTag* (*SimpleTag* ou *Tag*)

- Il faut par ailleurs modifier le fichier de descripteur de balises personnalisées

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

- Exemple complet montrant l'intérêt des attributs dynamiques
  - Un tag personnalisé remplaçant la balise <input> de type text
  - Associé les valeurs des champs avec des beans

```
<montagamoi:dynamicattribut type="text" name="nomDuBean" value="valeurDuBean" ... />
```

```
<input type='text' name='nomDuBean' value='valeurDuBean' ... />
```

  - La balise <input> accepte jusqu'à 12 attributs et 16 attributs d'événement
  - Sans les attributs dynamiques il faudrait les définir dans le TLD et ajouter les modifieurs dans la classe « handler »

www.serli.com
serli informatique

Attributs dynamiques (2.0) : exemple

- Exemple : puissances des attributs dynamiques

```
<%@ taglib uri="monTag2" prefix="montagamoi" %>
<jsp:useBean id="input3" scope="page" class="java.util.Date" />

Input 1 : <montagamoi:dynamicat type="text" name="input1" class="green" readonly="true" value="statique" /><br>
Input 2 : <montagamoi:dynamicat type="text" name="input2" class="red" onfocus="methodeJavaScript" /><br>
Input 3 : <montagamoi:dynamicat type="text" name="input3" class="blue" /><br>
```

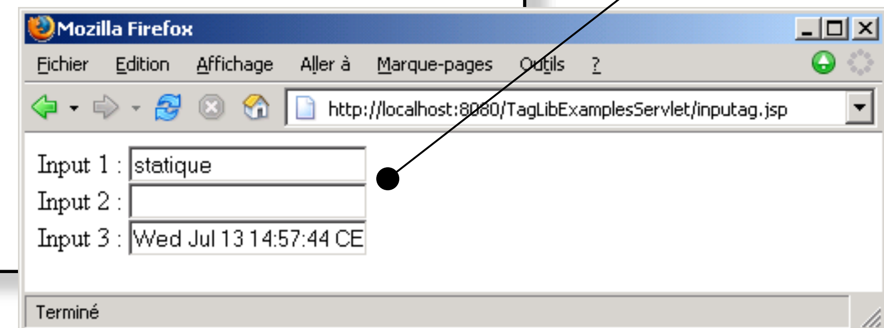
La page JSP

```
<tag>
    <name>dynamicattribut</name>
    <tag-class>monpackage.InputTag</tag-class>
    <description>Tag qui montre la déclaration d'une variable de script</description>
    <attribute>
        <name>name</name>
        <required>true</required>
        <rtexprvalue>true</rtexprvalue>
    </attribute>
    <attribute>
        <name>type</name>
        <required>true</required>
        <rtexprvalue>true</rtexprvalue>
    </attribute>
    <attribute>
        <name>value</name>
        <rtexprvalue>true</rtexprvalue>
        <required>false</required>
    </attribute>
    <body-content>scriptless</body-content>
    <dynamic-attributes>true</dynamic-attributes>
</tag>
```

Deux attributs sont obligatoires

Un attribut est obligatoire

La valeur « statique » et la valeur du Bean



Nous utilisons explicitement des attributs dynamiques

Attributs dynamiques (2.0) : exemple

- Exemple (suite) : puissances des attributs dynamiques

```
Input 1 : <input type='text' name='input1' value='statique' readonly='true' class='green' /><br>
Input 2 : <input type='text' name='input2' value='' class='red' onfocus='methodeJavaScript' /><br>
Input 3 : <input type='text' name='input3' value='Wed Jul 13 14:57:44 CEST 2005' class='blue' /><br>
```

Le source « html » après génération

```
public class InputTag extends SimpleTagSupport implements DynamicAttributes {
    private Map attributes = new HashMap();
    private String type = null;
    private String name = null;
    private String value = null;

    public void setDynamicAttribute(String uri, String localname, Object value)
        throws JspException {
        attributes.put(localname, value);
    }

    public void setName(String name) {
        this.name = name;
    }

    public void setType(String type) {
        this.type = type;
    }

    public void setValue(String value) {
        this.value = value;
    }
    ...
```

La méthode qui permet de gérer dynamiquement les attributs

Les modifieurs des attributs « statiques »

www.serli.com
serli informatique

Attributs dynamiques (2.0) : exemple

- Exemple (suite) : puissances des attributs dynamiques

Vérifie s'il est possible de récupérer l'instance du Bean donnée par « name » et stockée dans « value »

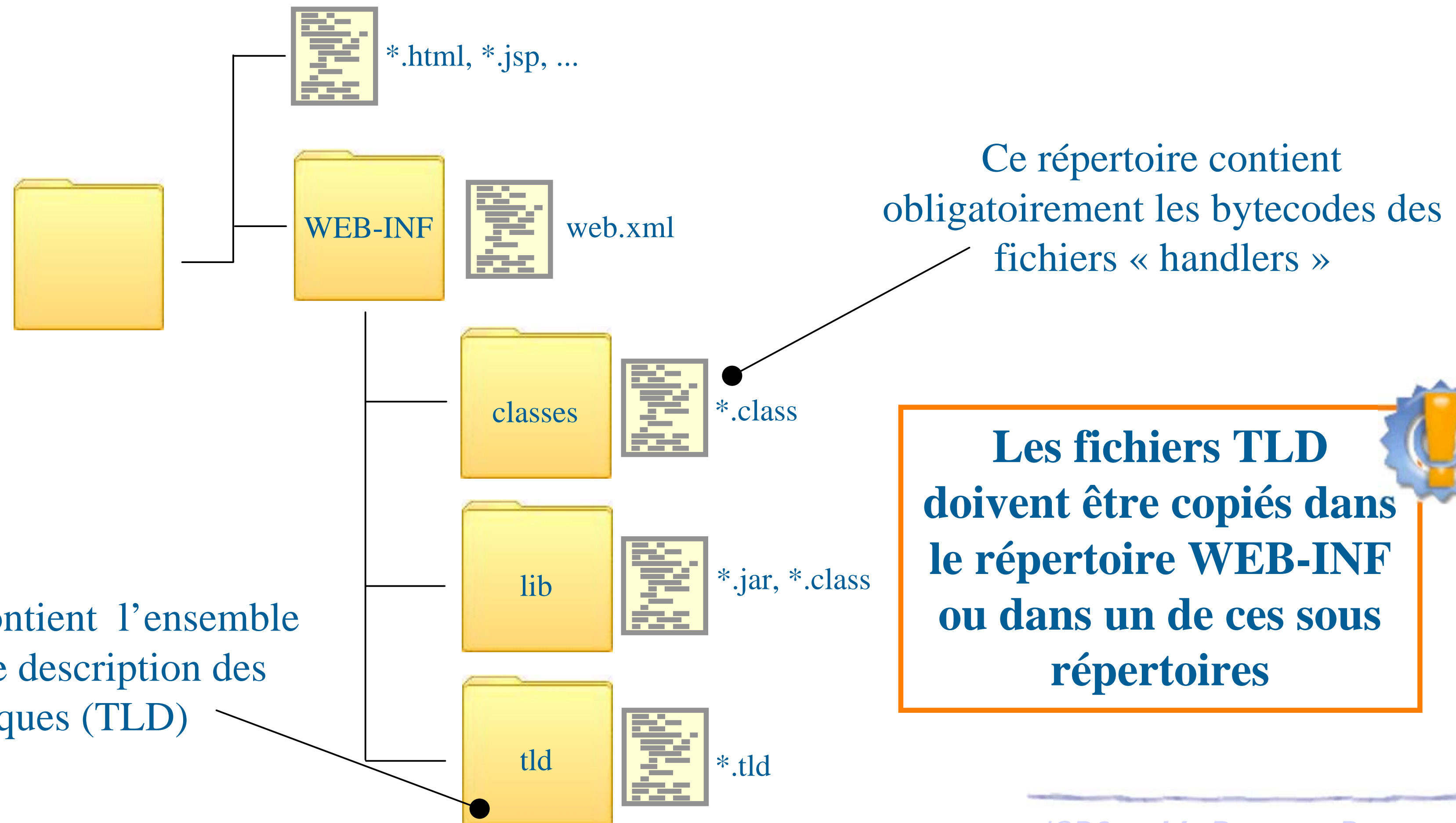
Gère les attributs statiques

```
    ...
    public void doTag() throws JspException, IOException {
            JspWriter out = getJspContext().getOut();
            out.print("<input type='" + type + "' name='" + name + "' ");
        if (value == null) {
            Object o = this.getJspContext().findAttribute(name);
            value = o == null ? "" : o.toString();
        }
        out.print("value='" + value + "' ");

        Iterator iterator = attributes.entrySet().iterator();
        while(iterator.hasNext()) {
            Map.Entry entry = (Map.Entry)iterator.next();
            out.print(entry.getKey() + "='" + entry.getValue() + "' ");
        }
        out.print("/>");
    }
}
```

Manipulation des attributs dynamiques et mise en forme

Déploiement dans une application WEB

- Il y a deux types d'éléments dont il faut s'assurer l'accès par le conteneur d'applications web (Tomcat en l'occurrence)
  - Les bytecodes des classes « handlers » des balises personnalisées
  - Les fichier de description des bibliothèques (TLD)



www.serli.com
serli informatique

Déploiement dans une application WEB

- Possibilité d'enregistrer les bibliothèques dans le fichier de configuration de l'application web (web.xml)
- Il faut ajouter dans le fichier web.xml, un tag *<taglib>* pour chaque bibliothèque utilisée contenant deux informations
  - l'URI de la bibliothèque *<taglib-uri>*
  - la localisation du fichier de description *<taglib-location>* relative au répertoire WEB-INF

```
...
<web-app ...>
    <display-name>
        Application WEB qui permet de gérer des Tags persos
    </display-name>
    <taglib>
        <taglib-uri>monTag</taglib-uri>
        <taglib-location>/WEB-INF/tld/montaglib.tld</taglib-location>
    </taglib>
</web-app>
```

JSTL (Java server page Standard Tag Library)

- Le but de la JSTL est de simplifier le travail des auteurs de JSP, c'est-à-dire les personnes responsables de la couche présentation

- La JSTL permet de développer des pages JSP en utilisant des balises XML sans connaissances du langage Java

- Sun a proposé une spécification pour la Java Standard Tag Library voir adresse *java.sun.com/jsp/jstl*

- L'implémentation proposée vient du projet Jakarta JSTL 1.1 disponible à l'adresse *jakarta.apache.org/taglibs*

www.serli.com
serli informatique

JSTL : Qu'est-ce-que c'est...

- C'est un ensemble de balises personnalisées qui propose des fonctionnalités souvent rencontrées dans les page JSP
  - Tag de structure (itération, conditionnement)
  - Internationalisation et formatage
  - Exécution de requête SQL
  - Utilisation de document XML

Nous ne présenterons que la partie liée aux structures et à l'internationalisation.

- JSTL 1.1 nécessite au minimum un conteneur JSP 2.0 (J2EE 1.4)
- Utilisation conjointe avec les Expressions Languages (EL)
- Possibilité également d'utiliser des scriplets mais non recommandées

www.serli.com
serli informatique

JSTL : la bibliothèque « Core »

- Cette bibliothèque « core »comporte les actions de base pour la gestion des variables de scope d'une application web
  - Affichage de variable, création, modification et suppression de variables de scope et de gestion des exceptions
  - Actions conditionnelles et Boucles
  - Manipulation d'URL, redirection
- Utilisation de la bibliothèque JSTL:core dans une application WEB
  - Copier *jstl.jar* et *standard.jar* dans le répertoire WEB-INF/lib
  - Copier le fichiers « c.tld » dans un sous répertoire de WEB-INF
  - Modifier le fichier « web.xml » de manière à enrichir l'URI

```
<taglib>
    <taglib-uri>cjstl</taglib-uri>
    <taglib-location>/WEB-INF/tld/c.tld</taglib-location>
</taglib>
```

  - Déclarer dans la page JSP la balise

```
<%@ taglib uri="cjstl" prefix= "c" %>
```

Dans la suite nous utiliserons le préfixe « c » pour utiliser cette bibliothèque

www.serli.com
serli informatique

JSTL : la bibliothèque « Core » : variables

- L'affichage d'une expression se fait par la balise *<out>*
- Les attributs de cette balise sont :
  - *Object value* : l'expression qui sera évaluée (obligatoire)
  - *Object default* : valeur à afficher si l'expression « value » est *null* (défaut : « »)
  - *boolean escapeXml* : détermines si les caractères <, >, &, ' et " doivent être remplacés par leurs codes respectifs
- Le corps du tag peut être utilisé à la place de l'attribut *default*

```
<c:out value="${ header[ 'user-agent']} " default="Inconnu" />
```

Corps de « out »

```
<c:out value="${header['user-agent']}" >
  default="Inconnu"
</c:out>
```

www.serli.com
serli informatique

JSTL : la bibliothèque « Core » : variables

- La balise personnalisée <*set*> permet de définir une variable de scope ou une propriété
- Les attributs de cette balise sont
  - *Object value* : l'expression à évaluer
  - *String var* : nom de l'attribut qui contiendra l'expression dans le scope
  - *String scope* : nom du scope qui contiendra l'attribut var (*page*, *request*, *session* ou *application*)
  - *Object target* : l'objet dont la propriété défini par « property » qui sera modifié
  - *String property* : nom de la propriété qui sera modifiée

```
<c:set var="variable" value="34" scope="page" />
```

- Le corps de la balise <*set*> peut être utilisé à la place de l'attribut « value »

www.serli.com
serli informatique

JSTL : la bibliothèque « Core » : variables

- Exemple : gestion des variables scopes

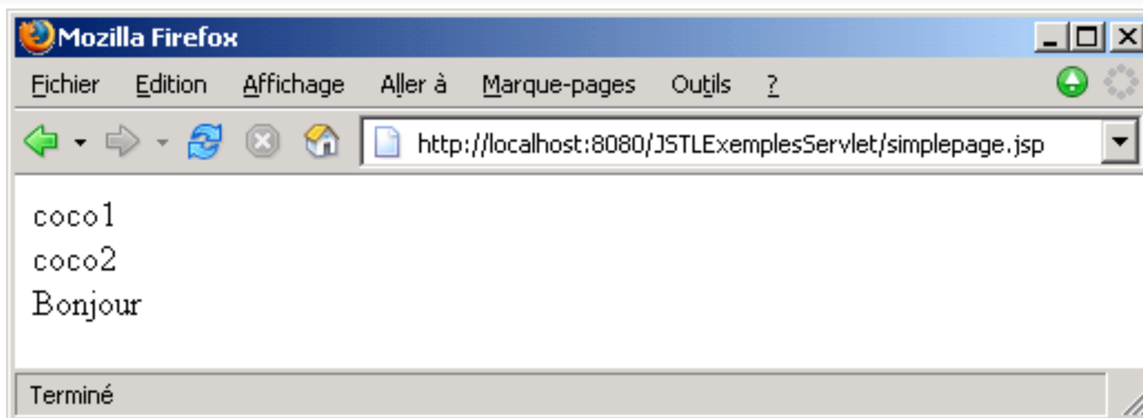
```
package monpackage;
public class MonBean {
    String name;
    public String getName() {

        return name;
    }
    public void setName(String p) {
        name = p;
    }
}
```

Fait la même chose de manière moins « programmation »

```
<%@ taglib uri="montag" prefix="c" %>

<c:set var="test1" value="coco1" scope="page"/>
<% pageContext.setAttribute("test2","coco2"); %>

<c:out value='${ test1} ' default='rien' /><br>
${ test2} <br>

<jsp:useBean id="monbean" class="monpackage.MonBean" scope="page" />
<c:set target="${ monbean} " value="Bonjour" property="name" />

<c:out value="${ monbean.name} " />
```

Affichage des informations

Création d'un objet *MonBean* et modification de sa propriété « name »



www.serli.com
serli informatique

JSTL : la bibliothèque « Core » : variables

- La balise personnalisée <*remove*> permet de supprimer la variable de scope indiquée

- Les attributs de cette balise sont
  - *String var* : nom de la variable de scope à supprimer (obligatoire)
  - *String scope* : nom du scope qui contiendra l'attribut « var » (*page*, *request*, *session* ou *application*)

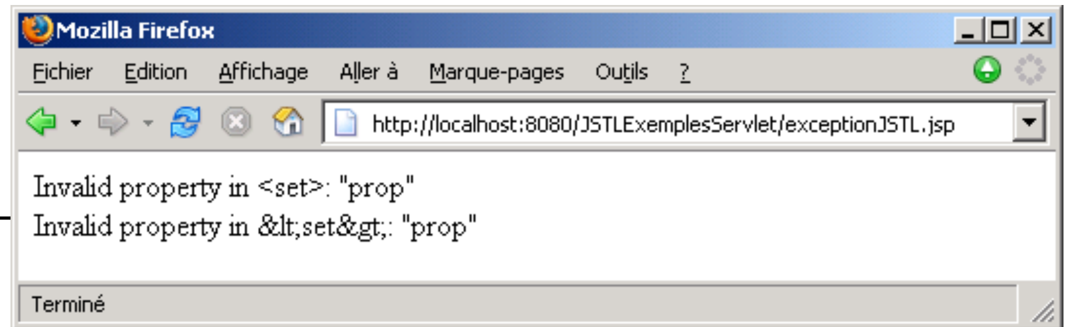
- Le corps de la balise <*remove*> ne contient aucune information

```
<c:remove var="test1" scope="page" />
```

www.serli.com
serli informatique

JSTL : la bibliothèque « Core » : variables

- La balise personnalisée <*catch*> permet d'intercepter les exceptions qui peuvent être lancées par son corps
- Le corps de la balise <*catch*> contient le code JSP a risque
- L'attribut de cette balise est
  - *String var* : nom de la variable dans le scope « page » qui contiendra l'exception interceptée
- Simple exemple :




```
<%@ taglib uri="montag" prefix="c" %>

<c:catch>
<c:set target="beans" property="prop" value="1" />
</c:catch>

<c:catch var="varName" >
<c:set target="beans" property="prop" value="1" />
</c:catch>

${varName.message} ou
<c:out value="${varName.message}" default="Rien" />
```

Le bean « beans » n'existe pas, l'exception est ignorée

Le bean « beans » n'existe pas, l'exception est stockée dans la variable « varname »

www.serli.com

JSTL : la bibliothèque « Core » : actions conditionnelles

- Le traitement conditionnel permet d'effectuer un traitement de la même manière que le mot-clef *if* du langage Java
- La balise personnalisée utilisée est <*if*>
- Elle dispose d'attributs qui sont
  - *boolean test* : condition qui détermine si le corps est évalué ou pas (obligatoire)
  - *String var* : nom d'une variable de type *boolean* contenant le résultat du test
  - *String scope* : valeur du scope pour l'attribut *var*

Si pas de paramètre dans la requête le corps n'est pas traité

```
<%@ taglib uri="montag" prefix="c" %>

<c:if test="${!empty param['page']}" var="valeur" scope="page">
Coucou ça fonctionne<br>
</c:if>

La valeur de la condition précédente est : ${valeur}
```

Possibilité d'afficher la valeur de la condition

JSTL : la bibliothèque « Core » : actions conditionnelles

- Le traitement conditionnel exclusif permet d'effectuer un traitement de la même manière que le mot-clef *switch* du langage Java
- La balise personnalisée utilisée est <*choose*>, elle ne dispose pas d'attribut et le corps peut comporter
  - une ou plusieurs balises de type <*when*>
  - zéro ou une balise <*otherwise*>
- L'action <*choose*> exécute le corps du premier tag <*when*> dont la condition de test est à *true*
- Si aucune des conditions n'est vérifiée, le corps de l'action <*otherwise*> est exécutée
- La balise <*when*> dispose d'un attribut
  - *boolean test* : détermine si le corps doit être évalué ou non (obligatoire)
- La balise <*otherwise*> ne dispose pas d'attribut

www.serli.com
serli informatique

JSTL : la bibliothèque « Core » : actions conditionnelles

- Exemple : traitement conditionnel exclusif

Définition d'une variable de script (scope à « page »)

```
<c:set var="value" value="2" scope="page" />

<c:choose>
   <c:when test="${ value==1} $">value vaut 1 (Un) </c:when>
   <c:when test="${ value==2} $">value vaut 2 (Un) </c:when>
   <c:when test="${ value==3} $">value vaut 3 (Un) </c:when>
   <c:when test="${ value==4} $">value vaut 4 (Un) </c:when>
   <c:otherwise>
      value vaut ${ value}
   </c:otherwise>
</c:choose>
```

Traitement conditionnel exclusif

Si pas de valeur commune message transmis par la balise < *otherwise*>

Pas d'empilage comme le *switch* du langage Java (utilisation du break)

www.serli.com

serli informatique

JSTL : la bibliothèque « Core » : boucles

- Le traitement des itérations permet d'effectuer un traitement itératif de la même manière que le mot-clef *for* et *while* du langage Java
- Deux balises sont définies *<forEach>* et *<forTokens>*
- Elles disposent en commun des attributs suivant
  - *String var* : variable qui comporte l'élément courant de l'itération
  - *String varStatus* : variable qui contient des informations sur l'itération
  - *int begin* : spécifie l'index de départ de l'itération
  - *int end* : spécifie l'index de fin de l'itération
  - *int step* : l'itération s'effectue sur les *step* éléments de la collection

www.serli.com
serli informatique

JSTL : la bibliothèque « Core » : boucles

- La balise <*forEach*> permet d'effectuer simplement des itérations sur plusieurs types de collection de données
- Elle dispose d'un attribut
  - *Object items* : collection d'éléments sur qui contient les éléments de l'itération
- L'attribut *items* accepte les éléments suivant
  - les tableaux d'objets ou de types primaires
  - les objets de type *Collection*
  - les objets de type *Iterator*
  - les objets de type *Enumeration*
  - les objets de type *Map*
- Une value *null* est considérée comme une collection vide
- Si l'attributs *items* est absent, *begin* et *end* permettent d'effectuer une itération entre deux nombres entiers

www.serli.com
serli informatique

JSTL : la bibliothèque « Core » : boucles

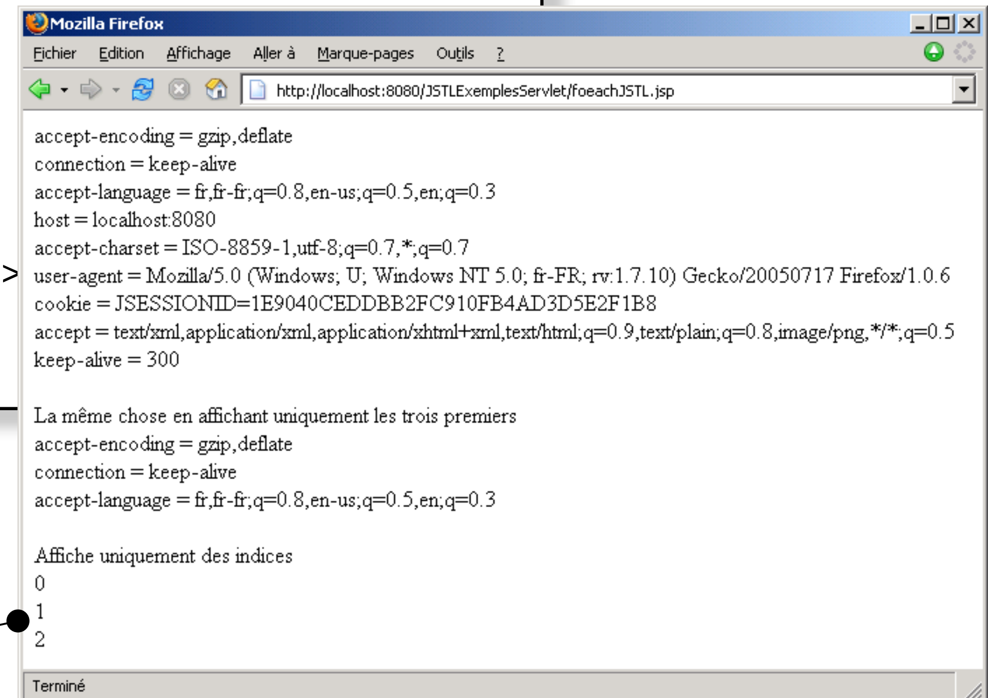
➤ Exemple : itérer sur une Collection

```
<%@ taglib uri="montag" prefix="c" %>

<c:forEach var="entry" items="${header}" >
    ${entry.key} = ${entry.value}<br>
</c:forEach>

<br>
La même chose en affichant uniquement les trois premiers<br>
<c:forEach var="entry" items="${header}" begin="0" end="2" >
    ${entry.key} = ${entry.value}<br>
</c:forEach>

<br>
Affiche uniquement des indices<br>
<c:forEach var="entry" begin="0" end="2" >
    ${entry}<br>
</c:forEach>
```

Affiche tous les éléments

Affiche uniquement les trois premiers éléments

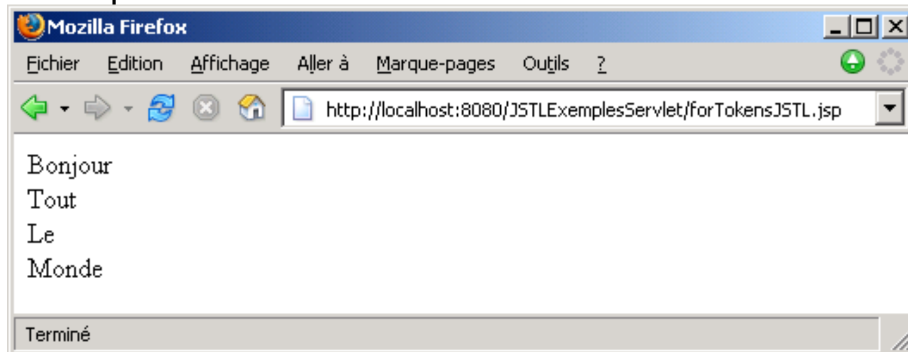
Affiche les nombres de 0 à 2



JSTL : la bibliothèque « Core » : boucles

- La balise *<forTokens>* permet de découper des chaînes de caractères selon un ou plusieurs délimiteurs
- Elle dispose d'un attribut
  - *String items* : collection d'éléments sur qui contient les éléments de l'itération (obligatoire)
  - *String delims* : la liste des caractères qui serviront de délimiteurs (obligatoire)
- Le corps de cette balise contient le code qui sera évalué pour chaque marqueur de chaîne

```
<%@ taglib uri="montag" prefix="c" %>

<c:forTokens var="content" items="Bonjour Tout Le Monde" delims=" ">
    ${content}<br/>
</c:forTokens>
```



www.serli.com
serli informatique

JSTL : la bibliothèque « Core » : URL

- Pour créer des URL's absolues, relatives au contexte, ou relatives à un autre contexte utiliser la balise <*url*>
- Cette balise contient différents attributs
  - *String value* : l'URL à traiter (obligatoire)
  - *String context* : spécifie le chemin du contexte de l'application locale
  - *String var* : le nom de la variable scope qui contiendra l'URL
  - *String scope* : nom du scope

Si *var* est omis, l'URL sera affichée dans la réponse

- Le corps de la balise peut contenir n'importe quel code JSP
- La balise <*param*> permet d'ajouter simplement un paramètre à une URL représentée par la balise parente
- Cette balise contient différents attributs
  - *String name* : nom du paramètre de l'URL (obligatoire)
  - *String value* : valeur du paramètre de l'URL
- Le corps peut-être utilisé à la place de *value*

JSTL : la bibliothèque « Core » : URL

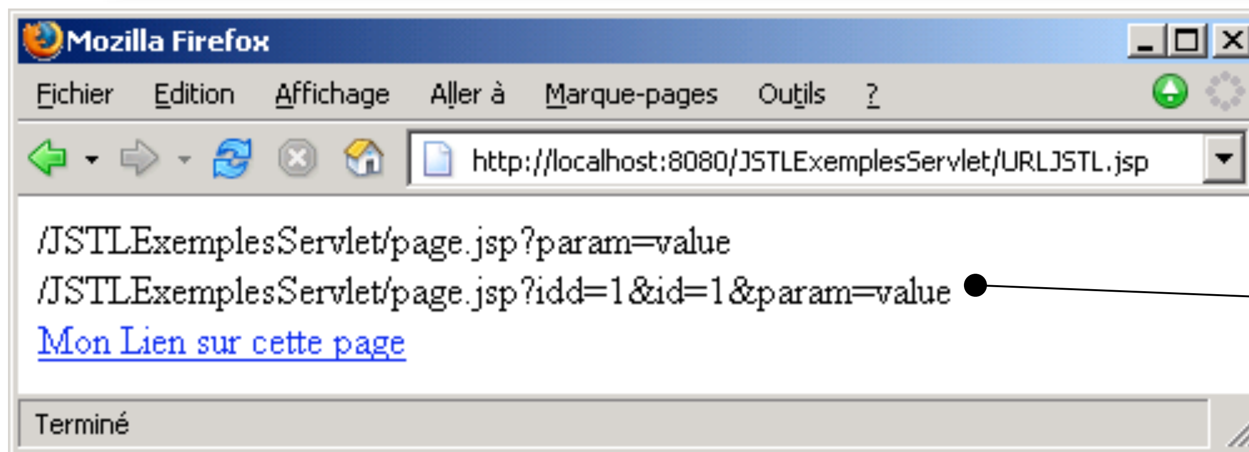
Exemple : écriture d'URL

Création d'une URL sans l'attribut *var* donc affichage dans la réponse

```
<%@ taglib uri="montag" prefix="c" %>

<c:url value="/page.jsp?param=value" /><br>

<c:url var="url" scope="page" value="/page.jsp?param=value" >
<c:param name="id" value="1"/>
</c:url>

<c:url var="url2" scope="page" context="/" value="${url}" >
<c:param name="idd" value="1"/>
</c:url>

${url2}<br>
<a href="${url}">Mon Lien sur cette page</a>
```

Création d'une URL avec ajout de paramètre

Création/modification d'une URL avec ajout d'un troisième paramètres



Ajoute le contexte à l'URL

www.serli.com
serli informatique

JSTL : la bibliothèque « Core » : URL

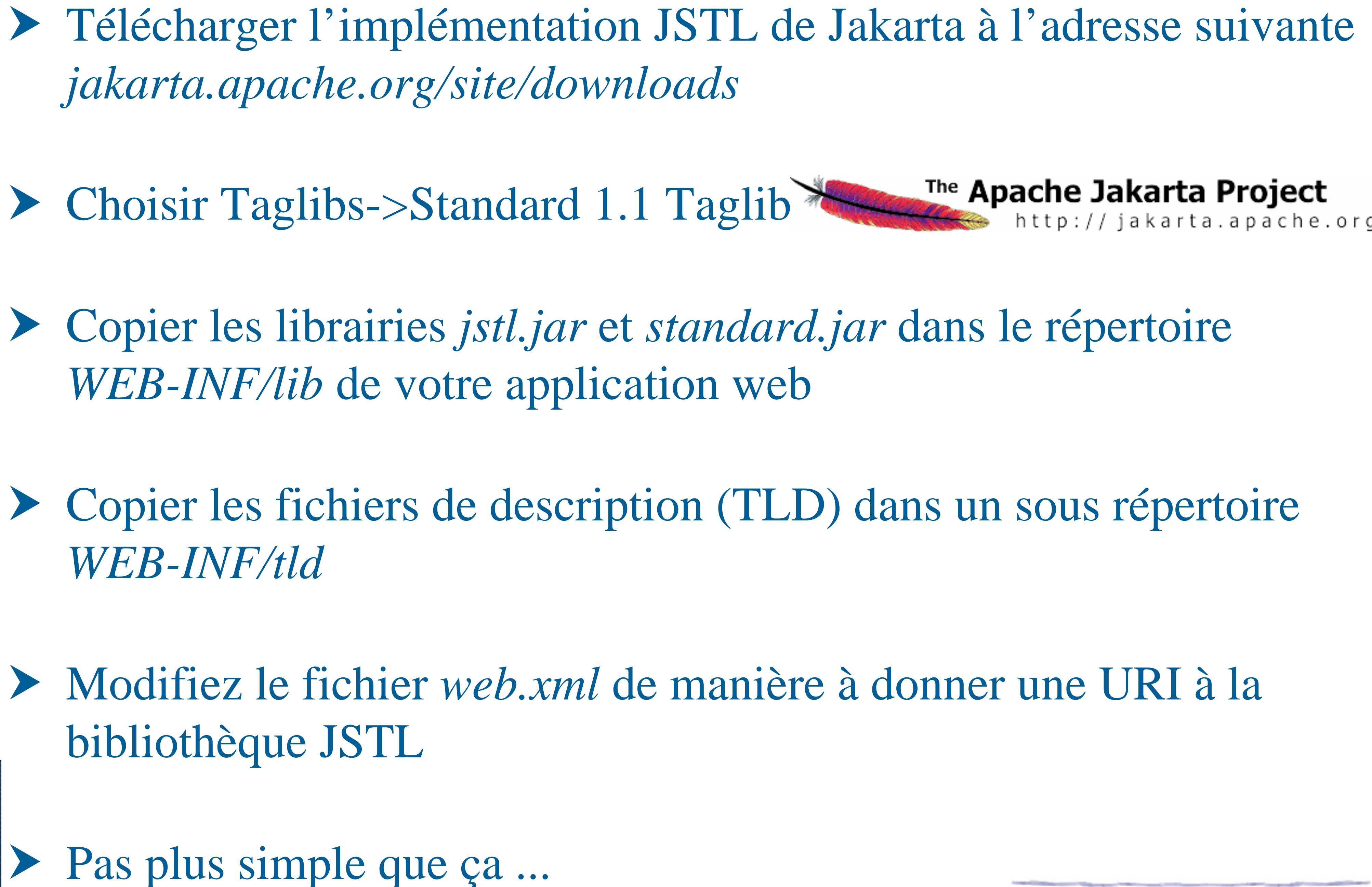
- La balise <*redirect*> est une commande de redirection HTTP au client
- Les attributs de cette balise sont
  - *String url* : l'url de redirection (obligatoire)
  - *String context* : spécifie le chemin du contexte de l'application locale à utiliser (débute obligatoirement par « / »)
- Le corps de la balise peut contenir n'importe quel code JSP

```
<%@ taglib uri="montag" prefix="c" %>

<c:redirect url="http://www.developpez.com" />

<c:redirect url="/exemple" >
   <c:param name="from" value="bonjour" />
</c:redirect>
```

Redirection à cette adresse

Cette URL ne sera jamais traité

JSTL : la bibliothèque « Core » : déploiement



www.serli.com
serli informatique

JSTL : la bibliothèque « I18n »

- La bibliothèque I18n facilite l'internationalisation d'une page JSP
  - Définition d'une langue
  - Formatage de messages
  - Formatage de dates et nombres
- Utilisation de la bibliothèque JSTL:fmt dans une application WEB
  - Copier *jstl.jar* et *standard.jar* dans le répertoire WEB-INF/lib
  - Copier le fichier « fmt.tld » dans un sous répertoire de WEB-INF
  - Modifier le fichier « web.xml » de manière à enrichir l'URI

```
<taglib>
    <taglib-uri>fmtjstl</taglib-uri>
    <taglib-location>/WEB-INF/tld/fmt.tld</taglib-location>
</taglib>
```

  - Déclarer dans la page JSP la balise

```
<%@ taglib uri="fmtjstl" prefix= "fmt" %>
```

Dans la suite nous utiliserons le préfixe « fmt » pour utiliser cette bibliothèque

JSTL : la bibliothèque « I18n »

- Les fichiers « properties » contenant les ressources doivent être placés dans le répertoire WEB-INF/classes
- La balise <*message*> permet de localiser un message
  - *String key* : clé du message à utiliser
  - *String var* : nom de la variable qui va recevoir le résultat
  - *String scope* : portée de la variable qui va recevoir le résultat
- Cette balise doit dépendre d'un fichier ressource (Bundle)
- La balise <*bundle*> permet de préciser un bundle à utiliser dans les traitements contenus dans son corps
  - *String basename* : nom de base de ressource à utiliser (obligatoire)

```
<fmt:bundle basename="message" >
    <fmt:message key="message.bonjour" />
</fmt:bundle>
```

Le message est affiché selon le bundle « message »

www.serli.com
serli informatique

JSTL : la bibliothèque « I18n »

- La balise <*bundle*> ne permet d'associer uniquement un bundle au traitement de son corps.

- La balise <*setBundle*> permet de forcer le bundle à utiliser par défaut
  - *String basename* : nom de base de ressource à utiliser (obligatoire)
  - *String var* : nom de la variable qui va stocker le nouveau bundle
  - *String scope* : portée de la variable qui va recevoir le nouveau bundle

Modification du bundle par défaut

```
<fmt:setbundle basename="message" />
...
<fmt:message key="message.bonjour" />
```

Le message est affiché par l'intermédiaire du bundle par défaut

Si le bundle n'est pas défini c'est celui par défaut qui est utilisé càd *null*

www.serli.com
serli informatique

JSTL : la bibliothèque « I18n »

- La balise <*setLocale*> permet de modifier une nouvelle locale
  - *String* ou *Locale value* : la locale à utiliser (obligatoire)
  - *String variant* : spécifie une variante spécifique à un système ou navigateur
  - *String scope* : nom du scope qui contiendra la locale

```
<fmt:setLocale value="en" scope="page" />
<fmt:setLocale value="fr" scope="session" />
```

Modification de la « locale » pour la page en cours

Modification de la « locale » pour l'utilisateur en cours

Si vous changer la locale dans le scope application cela affectera tous les utilisateurs

Possibilité de modifier plusieurs fois la locale dans une page JSP

www.serli.com
serli informatique

JSTL : la bibliothèque « I18n »

- La balise <*param*> permet de paramétrer l'affichage d'un message obtenu avec la balise <*message*>
- Cette balise n'est donc utilisable que dans la balise <*message*>
  - *Object value* : l'objet qui sera utilisé pour paramétrer le message
- Possibilité de mettre plusieurs paramètres pour un message
- Les paramètres sont utilisés dans le fichier properties en utilisant les accolades

```
<fmt:message key="la_cle">
    <fmt:param value="valeur1" />
    <fmt:param value="valeur 2" />
</fmt:message>
```

Ce message possède deux paramètres

Le fichier properties

```
la_cle=message {0}{1}
```

Les paramètres sont identifiés dans le fichiers properties par les accolades

www.serli.com
serli informatique

JSTL : la bibliothèque « I18n »

Exemple : gestion de l'internationalisation (message et date)

```
<%@ taglib uri="montag" prefix="c" %>
<%@ taglib uri="fmtjstl" prefix="fmt" %>

<fmt:setLocale value="en" scope="session" />

Un message de bienvenue en Anglais.
<fmt:bundle basename="informations" >
    <fmt:message key="bonjour" />
</fmt:bundle>
<br>
<fmt:setLocale value="fr" scope="session" />

<fmt:setBundle basename="message" />
Ce message ne sera pas traduit.<br>
<fmt:message key="message.vide" /><br>

<fmt:message key="autremessage.bonjour">
    <fmt:param value="Monsieur Patate" />
    <fmt:param value="Madame Patate" />
</fmt:message>.
```

La locale est fixée en anglais

Affichage de message à partir de la ressource « informations »

La locale est maintenant modifiée en français

Affichage de messages avec des paramètres

```
# Des Messages Rien que Des Messages
message.vide=Je pense que c'est vide
message.plein=Je pense que c'est plein

autremessage.bonjour=Bonjour {0} et {1}
autremessage.adieu=Bonsoir {0} et {1}
```

Fichier « message_fr.properties »