Java pour le développement d'applications Web : J2EE

Balises personnalisées

Mickaël BARON - 2006
(mailto:baron.mickael@gmail.com)

Bibliothèques de balises personnalisées : motivations ...

- Dans les pages JSP, on remarque qu'il y a mélange entre le code Java et le code HTML

- Le développeur a tendance à mettre trop de code dans un document qui ne devrait que fournir de l'affichage (HTML)

- Difficile à maintenir des pages JSP par un non programmeur
  - Concepteur de site WEB (non spécialiste des concepts objets)
  - Ne se charge que de la partie présentation

- Il faut donc fournir des outils qui encapsulent le code Java

- La solution
  - Cacher le code Java dans des balises personnalisées

Cette partie requiert des connaissances dans le langage XML

Bibliothèques de balises personnalisées

- Balises personnalisées aussi appelées :
  - actions personnalisées
  - tag lib (tag library)
  - tag personnalisés
- Élément du langage JSP défini par un développeur et non traités en standard par les JSP
- Permettent de définir ces propres balises qui réaliseront des actions pour générer la réponse
- Favorise la séparation des rôles entre les développeurs Java et les concepteurs de pages web
- Pour de plus amples informations sur les tags lib, le site de Sun *java.sun.com/products/jsp/taglibraries.html*

Bibliothèques de balises personnalisées

- Les balises personnalisées sont adaptées pour supprimer du code Java inclus dans les JSP est le déporter dans une classe dédiée
- La classe dédiée est comparable à un Java Bean qui implémente une interface particulière
- La différence étant qu'une balise personnalisée tient compte de l'environnement dans lequel il s'exécute et interagit avec lui
- Caractéristiques intéressantes des tags :
  - accès aux objets de la JSP (*HttpResponse*)
  - peuvent recevoir des paramètres envoyés à partir de la JSP
  - peuvent avoir un corps qu'ils manipulent ou pas

Qu'est-ce qu'un tag JSP

- Un tag JSP est une simple balise XML associée à une classe Java
- A la compilation d'une JSP, les balises sont remplacées par le résultat des classes Java associés aux balises
- Implicitement nous avons déjà utilisé dans des JSP des balises personnalisés au niveau des tags « action »

Balise ouvrante

```
<jsp:useBean id="monBean" class="Package.MonObjet" scope="atribut" >
    <jsp:setProperty name="monBean" property="*" />
</jsp:useBean>
```

Balise fermante

```
<jsp:forward page="/page.jsp" />
    <jsp:param name="defaultparam" value="nouvelle" />
</jsp:forward>
```

Attributs présents

L'utilisation des balises personnalisées permettent de limiter la présence de code Java

Qu'est-ce qu'un tag JSP

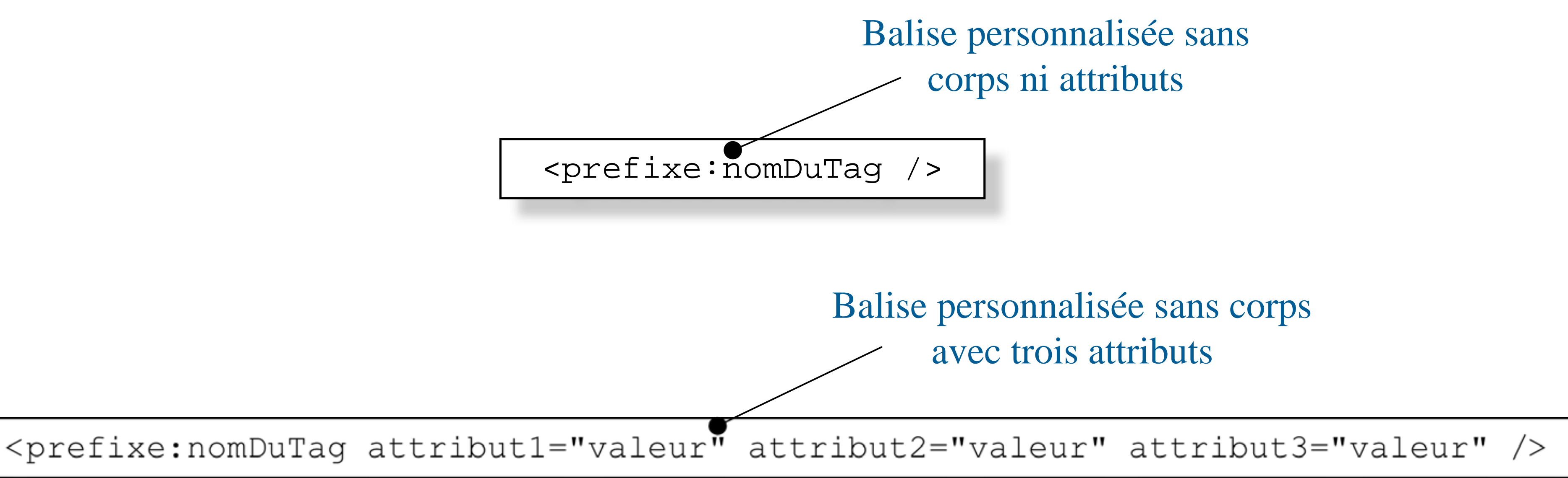
- La syntaxe d'un tag JSP est variable selon s'il dispose d'attributs ou de corps

```
<prefixe:nomDuTag attribut1="valeur" attribut2="valeur" >
    Corps du Tag
</prefix:nomDuTag>
```

- Nous retrouvons les éléments suivants :
  - *préfixe* : permet de distinguer les différents tag utilisés
  - *nomDuTag* : identifie le nom du tag de la librairie « préfixe »
  - Un certain nombre de couple d'attribut/valeur (peut-être au nombre de zéro)
  - Un corps (peut ne pas exister)

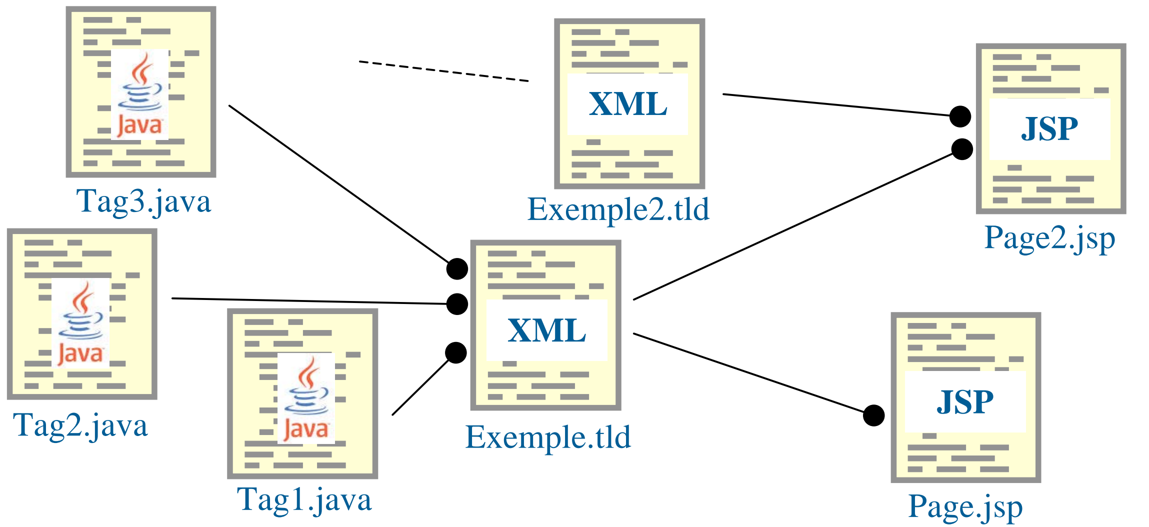
Qu'est-ce qu'un tag JSP

- Exemple : différentes formes de balises JSP (syntaxe XML)



Qu'est-ce qu'un tag JSP

- Un tag personnalisé est composé de trois éléments :
  - Tag Library Descriptor (TLD) ou description de la bibliothèque de balises effectue la relation (mapping) entre les balises et les classes Java (obligatoire)
    - Fichier de type XML
    - Le format porte obligatoirement l'extension « tld »
  - Une classe appelée « handler » pour chaque tag qui compose la bibliothèque (obligatoire)
  - Une classe permettant de fournir des informations supplémentaires sur la balise personnalisée au moment de la compilation de la JSP (facultatif)



Ces trois éléments seront étudiés plus en détails dans la sous partie liée à la conception d'une balise personnalisée

Utilisation dans une page JSP

- Pour chaque bibliothèque de balise à utiliser dans une JSP, il faut la déclarer avant en utilisant la directive *taglib*
  - *uri* : l'URI de la description de la bibliothèque (fichier *.tld)
  - *prefix* : espace de noms pour les tags de la bibliothèque dans la JSP

```
<%@ taglib uri="/taglib/mytag.tld" prefix="myprefix" %>
```

- Deux façons de définir l'uri :
  - directe (le nom du fichier TLD avec son chemin relatif)

```
<%@ taglib uri="/taglib/mytag.tld" prefix="myprefix" %>
```

  - indirecte (correspondance avec le fichier de description du contexte web.xml)

Détailler dans la partie liée au déploiement

```
<%@ taglib uri="myTagLib" prefix="myprefix" %>
```

```
<taglib>
    <taglib-uri>myTagLib</taglib-uri>
    <taglib-location>/taglib/mytag.tld</taglib-location>
</taglib>
```

Ajout dans le fichier web.xml

Conception d'un tag personnalisé

- Pour concevoir des balises personnalisées différentes versions existes. Les dernières versions supportent toujours les versions antérieures

- Le descripteur de bibliothèque de balise évolue en intégrant de nouvelles balises

- Implémentation de la classe « handler »
  - Version 1.2 : utilisation de l'interface *Tag*
  - Version 2.0 : utilisation de l'interface *SimpleTag*

JspTag << Interface >>

Tag << Interface >>

SimpleTag << Interface >>

- Pourquoi présenter les deux versions
  - Version 1.2 est toujours présentée et utilisée (livres, sites web, …)
  - Version 2.0 plus simple propose également les mêmes services que 1.2 (non dépréciés)

Conception d'un tag personnalisé (1.2)

- Évolutions vers la 1.2 depuis la 1.1
  - Normalisation des balises pour la description de la librairie de tag
  - Évolution du traitement du corps d'une balise personnalisée
- Les principales classes des balises personnalisées
  - *Tag* qui est l'interface de base pour écrire un tag
  - *BodyTag* une interface qui permet la gestion du corps d'un tag
  - *TagExtraInfo* apporte des informations complémentaires sur les tags
- Besoins de conception de deux familles d'élément
  - La classe « handler » qui implémente l'interface *Tag*
  - Le descripteur de la bibliothèque de tag (*.tld)

Conception d'un tag personnalisé par l'exemple (1.2)

Exemple : « HelloWorld » un classique

```
package monpackage;
...
public class HelloTag extends TagSupport {
    public int doStartTag() throws JspException {
          try {
            pageContext.getOut().println("Hello World !");
          } catch (IOException e) {
            throw new JspException ("I/O Error", e);
          }
          return SKIP_BODY;
    }
}
```

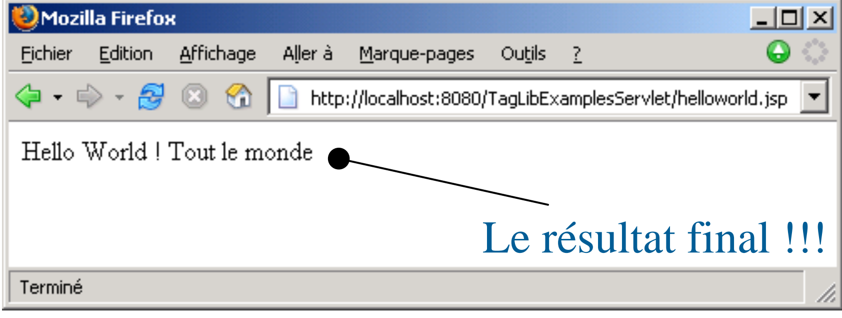
La classe « handler »

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<taglib ...>
     <tlib-version>1.0</tlib-version>
     <jsp-version>1.2</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



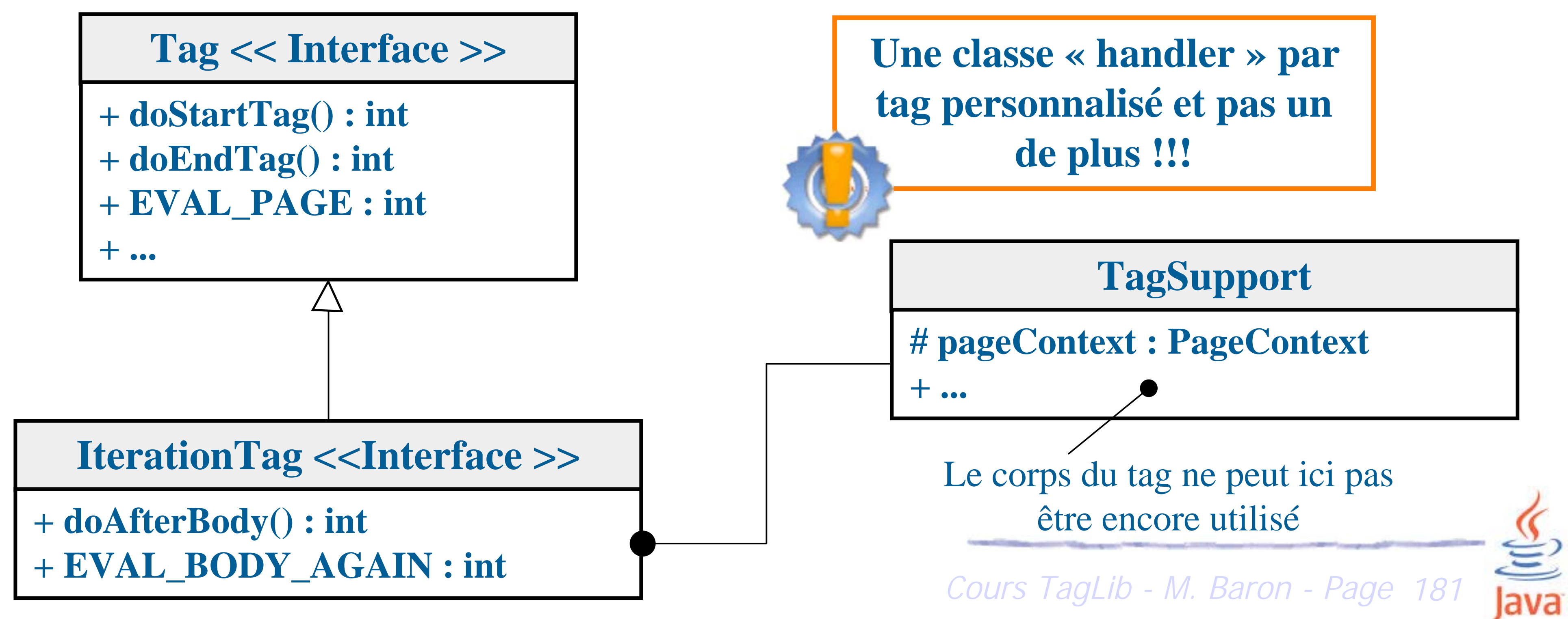
Le résultat final !!!

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

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

La JSP avec le nouveau Tag

Conception d'un tag personnalisé (1.2) : 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 *Tag*
- Préférez l'utilisation de la classe *TagSupport* qui implémente directement *Tag* (*javax.servlet.jsp.tagext.TagSupport*)



Conception d'un tag personnalisé (1.2) : cycle de vie (1)

- L'évaluation d'un tag JSP aboutit aux appels suivants
  - Initialisation de propriétés (*pageContext*, *parent)*
  - Initialisation des attributs s'ils existent
  - La méthode *doStartTag()* est appelée
  - Si la méthode *doStartTag()* retourne la valeur *EVAL_BODY_INCLUDE* le contenu du corps du tag est évalué
  - Lors de la fin du tag, la méthode *doEndTag()* est appelée
  - Si la méthode *doEndTag()* retourne la valeur *EVAL_PAGE* l'évaluation de la page se poursuit, si elle retourne la valeur *SKIP_PAGE* elle ne se poursuit pas
  - La méthode *release()* libère les ressources

TagSupport
+ doStartTag() : int **+ doEndTag() : int** **+ doAfterBody() : int** **+ getParent() : Tag** **+ setPageContext(PageContext)** **+ setParent(Tag)** **+ EVAL_BODY_AGAIN** **+ EVAL_BODY_INCLUDE** **+ EVAL_PAGE** **+ SKIP_BODY** **+ SKIP_PAGE** **# pageContext : PageContext** **...**

Conception d'un tag personnalisé (1.2) : « handler »

- La méthode *doStartTag()* est appelée lors de la rencontre de la balise d'ouverture, elle retourne un entier
  - EVAL_BODY_INCLUDE : poursuite du traitement avec évaluation du corps
  - SKIP_BODY : poursuite du traitement sans évaluation du corps
- La méthode *doEndTag()* est appelée lors de la rencontre de la balise de fermeture, elle retourne un entier
  - EVAL_PAGE : poursuite du traitement de la JSP
  - SKIP_PAGE : interrompre le traitement du reste de la JSP

```
package monpackage;
...
public class HelloTag extends TagSupport {
    public int doStartTag() throws JspException {
        try {
          pageContext.getOut().println("Hello World !");
        } catch (IOException e) {
          throw new JspException ("I/O Error", e);
        }
        return SKIP_BODY;
    }
}
```

Le traitement du tag est suivi par une non évaluation du corps

Conception d'un tag personnalisé (1.2) : 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 DTD
- La balise racine du document XML est la balise *<taglib>*
- En-tête du fichier TLD

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib
    PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.2//EN »
    "http://java.sun.com/dtds/web-jsptaglibrary_1_2.dtd">
<taglib>
    ...
</taglib>
```

Défini par un fichier DTD

Balise racine

Conception d'un tag personnalisé (1.2) : 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>1.2</jsp-version>
    <short-name>TagLibTest</short-name>
    <description>Bibliothèque de taglibs</description>
    <tag>
        ...
    </tag>
    <tag>
        ...
    </tag>
</taglib>
```

Première balise personnalisée

Seconde balise personnalisée

Conception d'un tag personnalisé (1.2) : TLD

- Chaque balise personnalisée est défini 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 (optionnelle)
    - **JSP** : le corps du tag contient des tags JSP
    - **tagdependent** : l'interpretation du contenu du corps est faite par le tag
    - **empty** : le corps doit obligatoirement être vide
  - *<description>* : description du tag (optionnelle)
  - *<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>
```

Chaque tag personnalisé est défini dans une balise <tag>

Conception d'un tag personnalisé (1.2) : 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 TagSupport {
	private Object attribut1;
	public void setAttribut1(Object p_attribut) {
		this.attribut1 = p_attribut;
	}
	...
}
```

- Des modifieurs prédéfinis sont utilisés pour initialiser des propriétés du tag (*pageContext* et *parent*)
  - *setPageContext(PageContext)*
  - *setParent(Tag)*

Conception d'un tag personnalisé (1.2) : 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 (1.2) : attributs de tag

Exemple 1 : « HelloWorld » avec des attributs

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

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

    public int doStartTag() throws JspException {
        try {
          pageContext.getOut().println ("Hello World ! " + moment);
        } catch (IOException e) {
          throw new JspException ("I/O Error", e);
        }
        return SKIP_BODY;
    }
}
```

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

Mozilla Firefox

Fichier Edition Affichage Aller à Marque-pages Outils ?

http://localhost:8080/TagLibExamplesServlet/h

Hello World ! Tout le monde
Hello World ! à toi

Terminé

Java

Conception d'un tag personnalisé (1.2) : attributs de tag

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

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<taglib ...>
    <tlibversion>1.0</tlibversion>
    <jspversion>1.2</jspversion>
    <description>Bibliothèque de test des taglibs</description>
    <tag>
        <name>hellotag</name>
        <tag-class>monpackage.HelloTag</tag-class>
        <description>Tag qui affiche bonjour</description>
    </tag>
    <tag>
        <name>hellotagattributs</name>
        <tag-class>monpackage.HelloTagAttributs</tag-class>
        <description>Affiche bonjour et un attribut</description>
        <attribute>
            <name>moment</name>
            <required>false</required>
            <rtexprvalue>false</rtexprvalue>
        </attribute>
    </tag>
</taglib>
```

Deux tags sont définis dans cette bibliothèque

Un seul attribut est défini

Conception d'un tag personnalisé (1.2) : 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

Conception d'un tag personnalisé (1.2) : attributs de tag

➤ Exemple 2 : cycle de vie avec *doStartTag()* et *doEndTag()*

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

    public int doStartTag() throws JspException {
        if (test.equals("body")) {
            return EVAL_BODY_INCLUDE;
        } else {
            return SKIP_BODY;
        }
    }

    public int doEndTag() throws JspException {
        if (apoca.equals("fin")) {
            return EVAL_PAGE;
        } else {
            return SKIP_PAGE;
        }
    }
}
```

```
<%@ 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.
```

Mozilla Firefox

Fichier Edition Affichage Aller à Marque-pages Outils ?

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

Le texte doit normalement s'afficher!!!

Terminé

Conception d'un tag personnalisé (1.2) : attributs de tag

➤ Exemple 3 : évaluation de code JSP depuis un attribut

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

    public int doStartTag() throws JspException {
        try {
            if (value instanceof Date) {
                pageContext.getOut().println((Date)value);
            } else {
                pageContext.getOut().println("Pas Objet Date");
            }
        } catch(IOException e) {
        }
        return TagSupport.SKIP_BODY;
    }
}
```

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

L'attribut peut recevoir une expression JSP

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

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

Un objet autre que *String* peut-être envoyé

Mozilla Firefox

Fichier Edition Affichage Aller à Marque-pages Outils ?

http://localhost:8080/TagLibExamplesServlet/testeva

Mon Jun 27 10:07:56 CEST 2005

Terminé

Conception d'un tag personnalisé (1.2) : variables implicites

- Les balises personnalisées ont accès aux variables implicites de la JSP dans laquelle ils s'exécutent via un objet de type *PageContext*
- Utilisation de l'attribut implicite *pageContext*
- La classe *PageContext* définit plusieurs méthodes
  - *JspWriter getOut()* : accès à la variable *out* de la JSP
  - *ServletRequest getRequest()* : accès à la variable *request*
  - *ServletContext getServletContext()* : accès à l'instance du ServletContext
  - *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, ...

Conception d'un tag personnalisé (1.2) : 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*

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

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

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

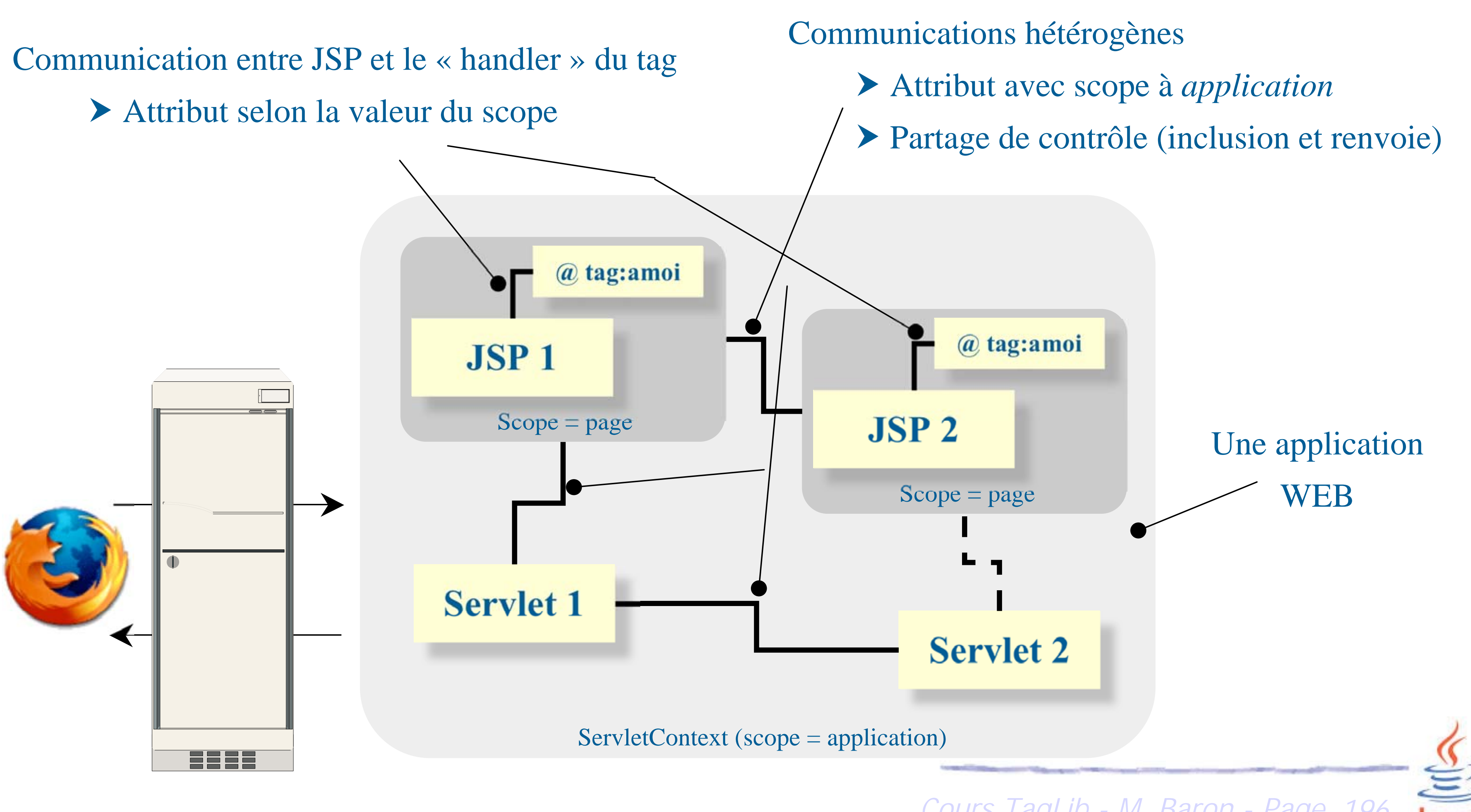
Cette méthode cherche l'attribut « toto » dans tous les scopes en commençant par *page*, *request*, *session* et *application*

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

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

Conception d'un tag personnalisé (1.2) : 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 (1.2) : 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 TagSupport {
    private String mon_bean;
    public void setName(String p_bean) {
        this.mon_bean = p_bean;
    }

    public int doStartTag() throws JspException {
    try {
        Object my_object = pageContext.findAttribute(mon_bean);
            if (my_object != null) {
                ArrayList my_array_list = (ArrayList)my_object;
                for (int i = 0; i < my_array_list.size(); i++) {
                pageContext.getOut().println(my_array_list.get(i));
                }
            } else {
                pageContext.getOut().println("Y a rien");
            }
        } catch (IOException e) {
            throw new JspException ("I/O Error", e);
        }
        return SKIP_BODY;
    }
}
```

Sachant que l'instance du Java Bean est dans le *PageContext*

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

Mozilla Firefox

Fichier Edition Affichage Aller à Marque-pages Outils ?

http://localhost:8080/TagLibExamplesServlet/helloworld2.jsp

[Fri Jun 24 15:15:59 CEST 2005, Fri Jun 24 15:16:00 CEST 2005, Fri Jun 24 15:16:01 CEST 2005, Fri Jun 24 15:18:56 CEST 2005, Fri Jun 24 16:40:24 CEST 2005] Fri Jun 24 15:15:59 CEST 2005 Fri Jun 24 15:16:00 CEST 2005 Fri Jun 24 15:16:01 CEST 2005 Fri Jun 24 15:18:56 CEST 2005 Fri Jun 24 16:40:24 CEST 2005

Terminé

Conception d'un tag personnalisé (1.2) : 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 int doStartTag() throws JspException {
        try {
            if (bean instanceof ArrayList) {
                if (bean != null) {
                    ArrayList my_array_list = (ArrayList)bean;
                    for (int i = 0; i < my_array_list.size(); i++) {
                        pageContext.getOut().println(my_array_list.get(i));
                    }
                } else {
                    pageContext.getOut().println("Y a rien");
                }
            }
        } catch(IOException e) {
        }
        return TagSupport.EVAL_BODY_INCLUDE;
    }
}
```

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

Conception d'un tag personnalisé (1.2) : variables implicites

Exemple 2 : traitement conditionnel du corps

```
public class ConditionelTraitementBody extends TagSupport {
    private String name = null;
    public void setName(String p_name) {
        this.name = p_name;
    }

    public int doStartTag() throws JspException {
        String value = pageContext.getRequest().getParameter(this.name);
        if (value != null) {
            return EVAL_BODY_INCLUDE;
        } else {
            return SKIP_BODY;
        }
    }
}
```

Évaluation du corps

Analyse la requête pour connaître le paramètre

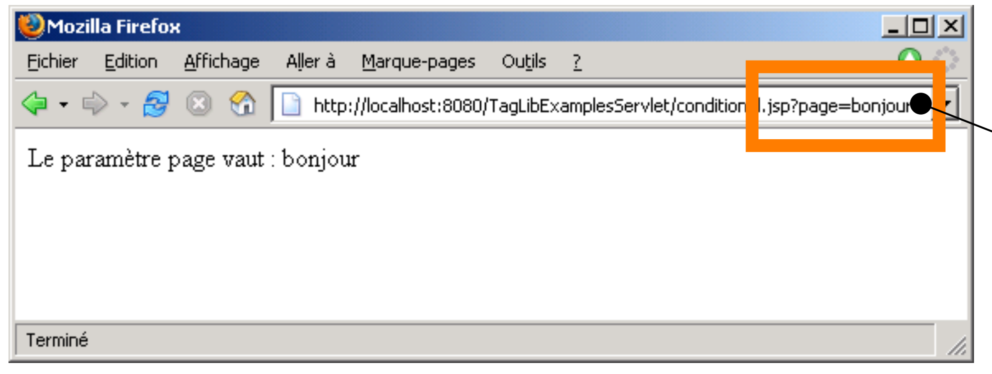
Non prise en compte du corps

L'attribut de cette balise JSP est obligatoire sinon exception

```
<tag>
<name>conditionelbody</name>
<tag-class>monpackage.ConditionelTraitementBody</tag-class>
<description>Tag qui affiche le corps ...</description>
<attribute>
    <name>name</name>
    <required>true</required>
    <rtexprvalue>true</rtexprvalue>
</attribute>
</tag>
```

Le tag expression est évalué

```
<%@ taglib uri="monTag" prefix="montagamoi" %>
<montagamoi:conditionelbody name="page" >
    Le paramètre page vaut : <%=request.getParameter("page")%>
</montagamoi:conditionelbody>
```



Le paramètre de la requête existe et le message est donc retourné

Conception d'un tag personnalisé (1.2) : variables implicites

Exemple 3 : 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 TagSupport {
    private String test;
    public int doStartTag() throws JspException {
        return EVAL_BODY_INCLUDE;
    }

    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 (1.2) : variables implicites

- Exemple 3 (suite) : collaboration de tag personnalisées ...

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

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

    public int doStartTag() throws JspException {
        if (this.getParent() instanceof SwitchTag) {
            SwitchTag parent = (SwitchTag)getParent();
            if (parent.isValid(this.value))
                return EVAL_BODY_INCLUDE;
            } else {
                return SKIP_BODY;
            }
        }
        throw new JspException("Le Tag 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

Mozilla Firefox

Fichier Edition Affichage Aller à Marque-page

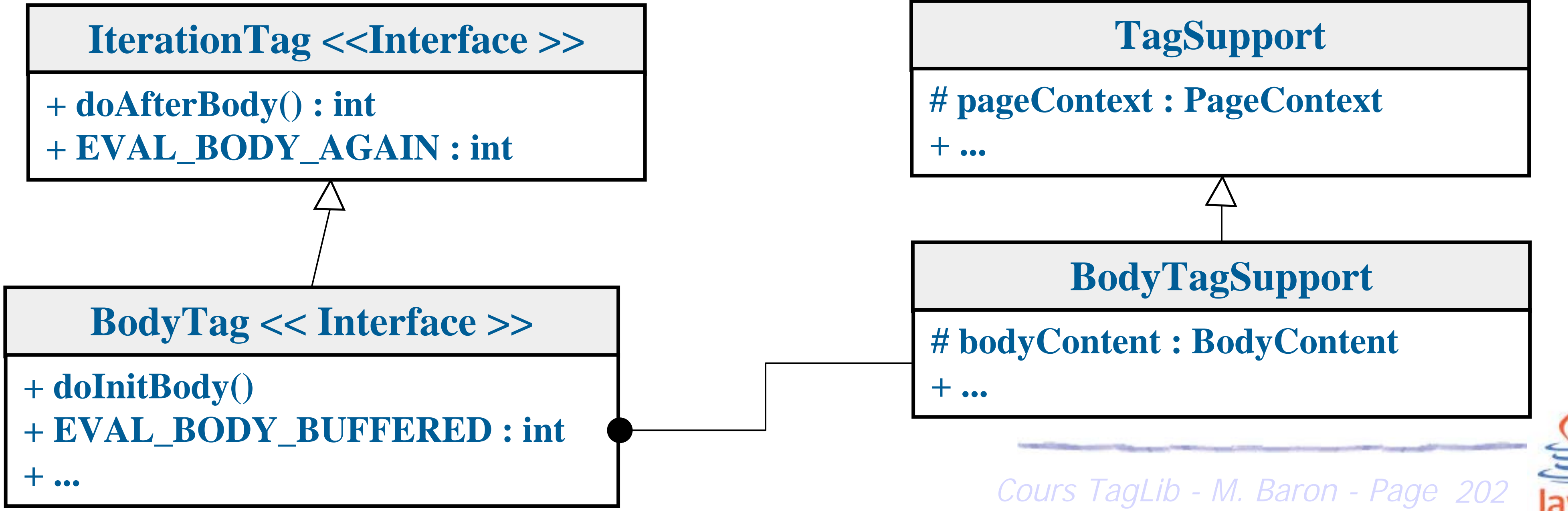
http://localho

Trois

Terminé

Conception d'un tag personnalisé (1.2) : interface BodyTag

- L'interface *BodyTag* étend l'interface *Tag*
- Elle permet le traitement **bufférisé** du corps de la balise ainsi que des **itérations** sur le corps du tag
- La valeur renvoyée par *doStartTag()* permet d'évaluer ou pas le corps du tag
- Préférez l'utilisation de la classe *BodyTagSupport* qui implémente directement *Tag* (*javax.servlet.jsp.tagext.BodyTagSupport*)



Conception d'un tag personnalisé (1.2) : cycle de vie (2)

- **Itération** sur le corps d'un tag JSP sans manipulation du contenu
  - Initialisation de propriétés (*pageContext*, *bodyContent)*
  - Initialisation des attributs s'ils existent
  - Si la méthode *doStartTag()* renvoie *EVAL_BODY_INCLUDE*
  - Évalue le contenu du corps de la balise
  - Boucle sur la méthode *doAfterBody()* si elle retourne *EVAL_BODY_AGAIN*
  - Appel de la méthode *doEndTag()* ...
  - La méthode *release()* libère les ressources

Aucune possibilité de manipuler le corps de la balise

L'itération est obtenue par l'interface *IterationTag* disponible également dans *TagSupport*

Conception d'un tag personnalisé (1.2) : cycle de vie (2)

- Traitement **bufférisé** du corps de tag JSP :
  - Initialisation de propriétés (*pageContext*, *bodyContent)*
  - Initialisation des attributs s'ils existent
  - Si la méthode *doStartTag()* renvoie *EVAL_BODY_BUFFERED*
  - La méthode *doInitBody()* est appelée
  - Évalue le contenu du corps de la balise
  - Boucle sur la méthode *doAfterBody()* si elle retourne *EVAL_BODY_AGAIN*
  - Appel de la méthode *doEndTag()* ...
  - La méthode *release()* libère les ressources

BodyTagSupport
+ doStartTag() : int **+ doEndTag() : int** **+ doAfterBody() : int** **+ doInitBody()** **+ getBodyContent() : BodyContent** **+ setBodyContent(BodyContent)** **+ getPreviousOut() : JspWriter** **+ EVAL_BODY_AGAIN** **+ EVAL_BODY_BUFFERED** **# pageContext : PageContext** **# bodyContent : BodyContent** **...**

Conception d'un tag personnalisé (1.2) : itération sur le corps

Exemple 1 : itération sur le corps du tag ...

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

    public void setCount(int i) {
        count = i;
    }
    public int doStartTag() throws JspException {
        current = 0;
        if (current < count) {
            return EVAL_BODY_INCLUDE;
        } else {
            return SKIP_BODY;
        }
    }
    public int doAfterBody() throws JspException {
        current++;

        if (current < count) {
            return EVAL_BODY_AGAIN;
        } else {
            return SKIP_BODY;
        }
    }
}
```

Appel la méthode *doAfterBody* sans modification du contenu

A chaque *doAfterBody* affichage du corps

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

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

Mozilla Firefox

Fichier Edition Affichage Aller à Marque-pages Outils

http://localhost:8080/Tag

Coucou
Coucou
Coucou

Terminé

Conception d'un tag personnalisé (1.2) : itération sur le corps

➤ Exemple 1 (bis) : itération sur le corps du tag avec modification

```
public class IterateTag extends BodyTagSupport {
	private int count = 0; private int current;
	public void setCount(int i) {
		count = i;
	}
	public int doStartTag() throws JspException {
		current = 0;
		if (current < count) {
			return EVAL_BODY_BUFFERED;
		} else {
			return SKIP_BODY;
		}
	}
	public void doInitBody() throws JspException {
		...
	}
	public int doAfterBody() throws JspException {
		try {
			current++;
			if (current < count) {
				return EVAL_BODY_AGAIN;
			} else {
				getBodyContent().getEnclosingWriter().println(getBodyContent().getString() + current);
				return SKIP_BODY;
			}
		} catch (IOException e) { ... }
		return SKIP_BODY;
	}
}
```

Appel la méthode *doStartTag* avec la possibilité de manipuler le contenu du corps

Traitement bufférisé donc le corps s'empile à chaque EVAL_BODY_AGAIN

Possibilité d'écrire sur la réponse

Accès au contenu bufférisé

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

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

Conception d'un tag personnalisé (1.2) : BodyTagSupport

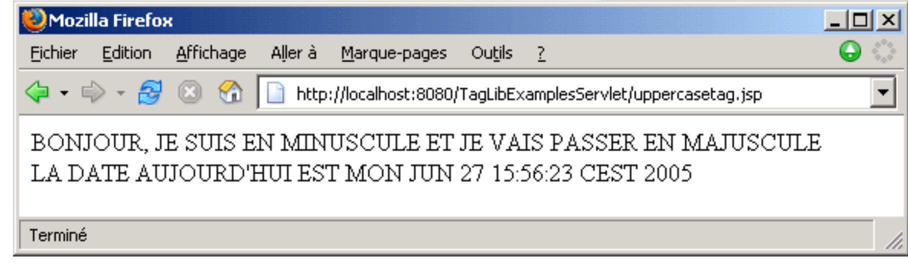
➤ Exemple 2 : modification du corps du tag

```
public class UpperCaseTag extends BodyTagSupport {
    public int doAfterBody() throws JspException {
        try {
            if (this.getBodyContent()!= null) {
                String body_string = getBodyContent().getString();
                body_string = body_string.toUpperCase();
                getBodyContent().getEnclosingWriter().println(body_string);
            }
        } catch(IOException e) {
            throw new JspException(e);
        }
        return EVAL_PAGE;
    }

    public int doStartTag() throws JspException {
        return EVAL_BODY_BUFFERED;
    }
}
```

Capture le contenu du *BodyContent* (bufférisé)

Affichage de la nouvelle valeur

```
<%@ 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>
```



Conception d'un tag personnalisé (1.2) : 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 des 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

Conception d'un tag personnalisé (1.2) : 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 int doStartTag() throws JspException {
        pageContext.setAttribute(name,new Date());
        return EVAL_BODY_INCLUDE;
    }

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

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

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

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

Utilisation obligatoire de l'objet implicite *pageContext* pour accéder au contenu de « value »

Conception d'un tag personnalisé (1.2) : 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 int doStartTag() throws JspException {
        pageContext.setAttribute(name,new Date());
        return EVAL_BODY_INCLUDE;
    }

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

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

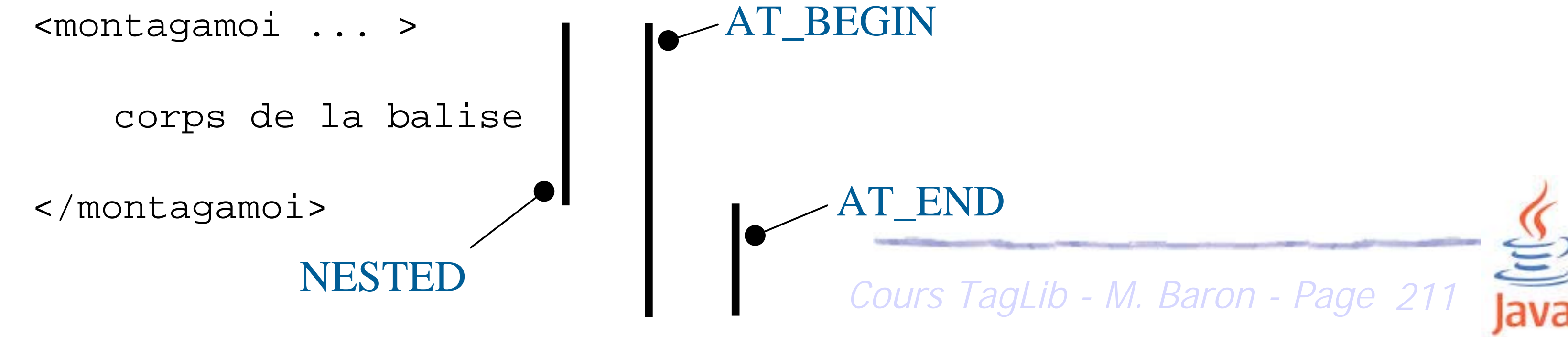
```
<%@ taglib uri="monTag" prefix="montagamoi" %>

<montagamoi:variablescript name="value" >
<%= value %><br>
</montagamoi:variablescript>

<%= value %>
```

Utilisation de la variable de script « value » librement

A suivre ...

Conception d'un tag personnalisé (1.2) : 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 (1.2) : 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

Conception d'un tag personnalisé (1.2) : 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

Conception d'un tag personnalisé (1.2) : 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 %>
```

Définition d'un attribut dans le scope « page »

Récupère l'intégralité des attributs du tag

```
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;
    }
}
```

« name »

« value »

Déclaration de la variable de script

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

Typage de la variable de script

Conception d'un tag personnalisé (1.2) : 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>
</tag>
```

Conception d'un tag personnalisé (1.2) : 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

Conception d'un tag personnalisé (1.2) : 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;
    }
```

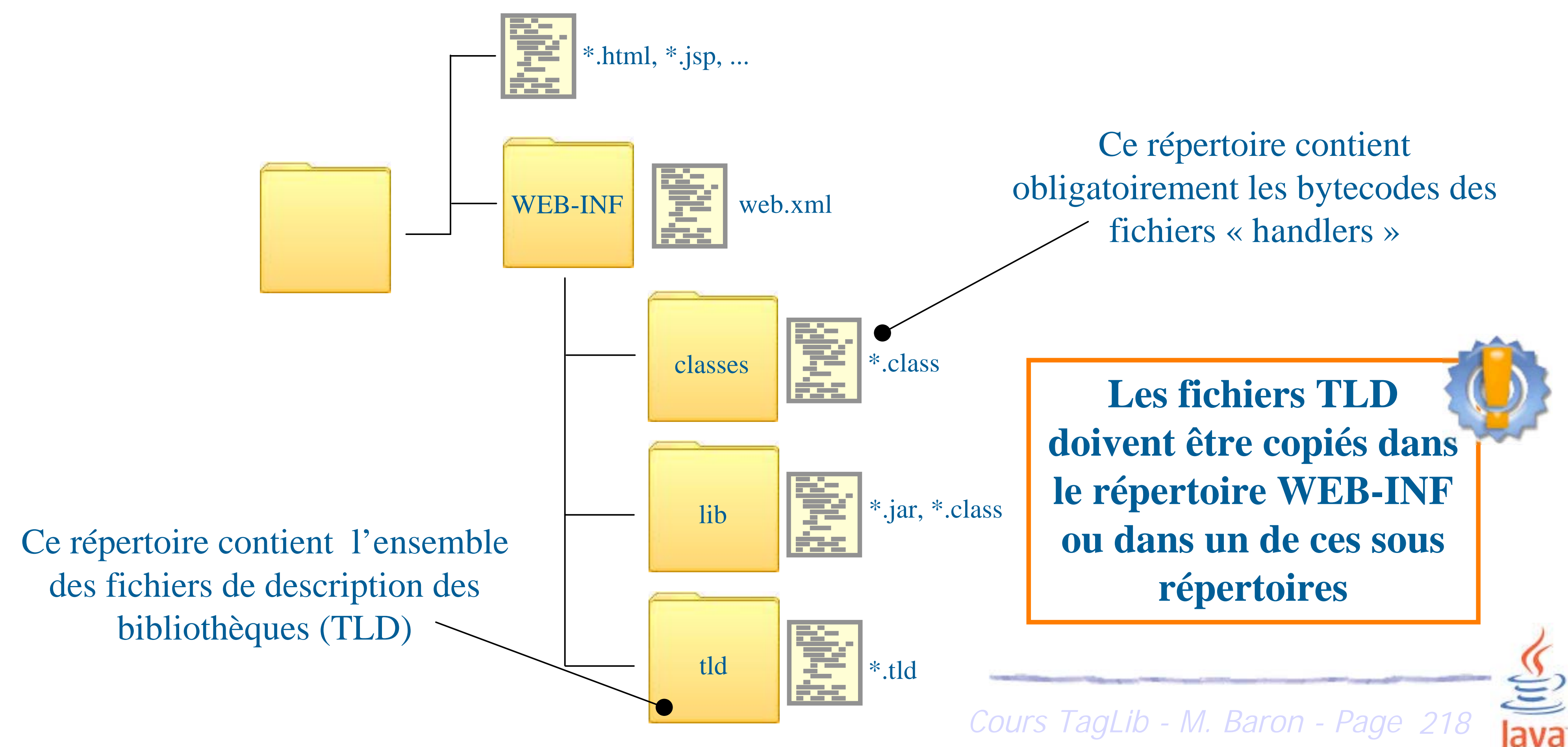
Affichage dans la console avant l'exécution de la page JSP

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

Impossibilité de vérifier le contenu d'un tag expression

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)



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>
```