Dans l'étude précédente, nous avons présenté les pages JSP et nous en avons suffisamment appris pour commencer à écrire des pages et à les utiliser pour construire des applications Web. Cette étude va nous aider à concevoir des pages JSP de façon beaucoup plus agréables. En effet, il faut reconnaître que pour l'instant, nous trouvons encore trop de code Java au milieu du balisage HTML et JSP.

Effectivement, il faut se mettre à la place du webmaster qui ne possède pas spécialement des connaissances en Java, mais par contre, il est particulièrement efficace dans la manipulation des balises. Nous allons faire en sorte d'aider au maximum le webmaster en lui proposant des éléments adaptés à la situation. Les techniques que nous allons mettre en oeuvre pour cela sont les suivantes :

1. *Langage d'expressions - permet de manipuler des données de façon plus naturelle. Ce langage permet, par exemple, l'accès aux propriétées des objets placés en attribut des scopes, aux paramètres de la requête sans passer par la syntaxe des expressions classiques de jSP <%= %> qui est un peu délicate à manipuler et d'une syntaxe toujours plus longue.*
2. *Actions personnalisées - Les actions standards permettent d'encapsuler le code Java de façon à ce qu'il soit invisible dans les pages JSP. Les développeurs des pages n'ont ainsi à connaître que la syntaxe des actions, et non la syntaxe Java. Mais les actions standards peuvent accomplir un nombre de tâches limité. Heureusement, il est possible de créer de nouvelles actions, appelées actions personnalisées. Les balises utilisées pour définir ces actions sont appelées balises étendues.*
3. *JSTL - Plutôt que des développeurs créent des actions personnalisées chacun de leur côté pour des tâches courantes, la spécification JSTL (JSP Standard Tag Library - bibliothèque standard de balises JSP) a été développée pour répondre à la plupart des besoins et procure déjà un grand nombre d'actions très utiles et très intéressantes.*

Avec les actions JSP, les actions JSTL, les actions personnalisées, les Javabeans, il est possible de construire un site n'utilisant qu'une syntaxe XML.

Langage d'expressions - EL

Dans l'étude précédente, nous avons vu comment créer des éléments de script pouvant être utilisés pour placer du code Java dans les pages JSP. Ces éléments sont les déclarations, les scriptlets et les expressions :

```
<%! int a = 5; %> <%-- déclaration --%>
<% while (x<10) { out.write("x="+x); } %> <%-- srciptlet --%>
<%= utilisateur.getPrénom() %> <%-- expression --%>
```

La spécification JSP 2.0 a ajouté le langage d'expressions (Expression Language, ou EL) à la panoplie des outils JSP. Les instructions de ce langage sont d'une syntaxe plus simple tout en permettant d'obtenir les mêmes résultats que les éléments de script.

Une expression en EL débute par ${ et se termine par } . La syntaxe fondamentale est donc la suivante :

```
${une expression}
```

L'expression ne s'imprime pas à l'écran sans être traitée. Ainsi :

```
${utilisateur}
```

peut être une variable par exemple. Mais elle peut être aussi le résultat d'une opération. Ainsi :

```
${1+2} imprime 3 sur la page HTML envoyée
```

Ce trois correspond d'ailleurs au caractère '3' ce qui veut dire que le changement de type est automatiquement réalisé. Par ailleurs, les caractères spéciaux sont interprétés comme du code HTML. Si une variable contient " é ", elle s'exprimera &eacute ; . Ainsi l'expression :

```
${prénom} (ou prénom est égal à "rené") imprime sur la page HTML " ren&eacute ; "
```

Les expressions peuvent comporter des constantes littérales, des opérateurs, des références vers des variables ou vers des objets implicites, et des appels de fonctions.

Les constantes littérales

La syntaxe EL comporte plusieurs types de constantes littérales qui peuvent être employées dans les expressions :

1. *Les valeur booléennes - true, false.*
2. *Les chaînes de caractères - N'importe quelle suite de caractères placées entre guillemets ou entre apostrophes. La barre oblique inverse peut être utilisée comme caractère d'échappement pour les guillemets et les apostrophes. Par exemple : 'Cette chaîne contient un exemple d\'apostrophe avec deux caractères d\'échappement' ou : "le dossier est c:\\Mes documents\\Travail". Le caractère d'échappement est uniquement nécessaire pour les caractères identiques aux délimiteurs de la chaîne. Du coup, puisque nous avons le choix, il suffit de prendre les bons délimiteurs. Le caractère d'échappement n'est pas nécessaire dans l'exemple suivant : "Cette chaîne contient un exemple d'apostrophe sans caractère d'échappement".*
3. *Les valeurs entières - N'importe quel nombre entier, positif ou négatif : -13, 45, 2374, etc.*
4. *Les valeurs réelles - N'importe quelle valeur numérique exprimée en virgule flottante (mais en remplaçant la virgule par un point !) : -1.3E-30, 3.14159, .45, etc.*
5. *La valeur null.*

En voici quelques exemples :

```
${true}
${"Les apostrophes à l'intérieur des guillemets n'ont pas besoin de caractères d'échappement"}
${2*4} --> 8
${(2*4)+3} --> 11
```

Les opérateurs

La plupart des opérateurs Java sont disponibles dans le langage d'expressions :

Type	Opérateurs
Arithmétique	+ - * / div % mod
Relationnels	== et eq != et ne < et lt > et gt <= et le >= et ge
Logiques	&& et and || et or ! et not
Autre	( ) empty [ ] .

La plupart des opérateurs nous sont familiers. Notez toutefois que de nombreux opérateurs disposent d'une notation symbolique et d'une variante littérale, tels / et div, ou && et and. Ces équivalents sont utiles si vos pages JSP doivent être conformes à la norme XML. Cela évite d'avoir à utiliser des entités, tel < à la place de <. Dans un document XML, une expression EL signifiant " plus petit que " peut être codé :

```
${2 lt 3} plutôt que ${2 < 3}
```

Voici quelques exemples :

```
${(2*4)+3} --> 11
${2*(3+4)} --> 14
${maListe[1] > monTableau[2] and test == false}
```

L'opérateur empty

L'opérateur empty permet de tester différentes conditions. L'expression :

```
${empty nom}
```

retourne true si nom est null, ou s'il référence une instance vide de String, List, Map, ou encore un tableau vide. Dans le cas contraire, empty retourne la valeur false. L'objet référencé par nom peut se trouver dans un des objets implicites page, request, session ou application. Par exemple :

```
<% Vector vec = new Vector();        // Crée un vecteur vide
        pageContext.setAttribute("nom", vec); %>// Place le vecteur dans pageContext
...
${empty nom}    // Evalué à true ; notez que l'opérateur prend pour opérande l'attribut nom, et non la valeur vec
```

1. *N'oubliez pas que cette technique fonctionne pour tous les objets présents dans un des contextes, et non pas seulement pour ceux qui y sont ajoutés explicitement, par exemple à l'aide de la méthode setAttribute( ).*
2. *Comme nous le verrons plus loin, les actions personnalisées peuvent créer des variables accessibles grâce à des expressions EL. L'opérateur empty peut être appliqué à ces variables.*
3. *Les JavaBeans sont également un moyen d'ajouter des objets aux contextes. Les JavaBeans sont stockés dans le contexte correspondant à l'attribut scope de l'action <jsp:useBean>. L'important est que l'opérateur empty peut être appliqué à tous les objets se trouvant sur un des contextes et pouvant être référencé par un nom.*

Les opérateurs . et [ ]

Les deux derniers opérateurs sont le point (.) et l'opérateur d'indexation ([ ]). Ils sont utilisés pour accéder aux attributs d'un objet de la page. La valeur située à gauche de l'opérateur est interprétée comme représentant un objet de la page. La valeur située à droite est une propriété, une clé, ou un index. Par exemple, si vous avez défini un bean à l'aide de l'action standard <jsp:useBean>, vous pouvez accéder à ses propriétés en utilisant ces deux notations. Si un bean utilisateur possède deux propriétés nom et prénom, celles-ci sont accessibles en utilisant les notations suivantes :

```
${utilisateur.prénom}
${utilisateur[prénom]}
```

Ces deux notations sont équivalents pour accéder aux propriétés d'un objet. Dans les deux cas, la page tente de trouver l'objet utilisateur et d'appeler sa méthode getPrenom( ).

Ces deux opérateurs peuvent également être employés avec des objets de type Map ou List, ou encore avec des tableaux. Si l'objet est une Map, la valeur de droite est utilisée comme une clé. ainsi le code suivant :

```
${unObjet[nom]} ou ${unObjet.nom}
```

est équivalent à :

```
unObjet.get("nom") ;
```

Si l'opérateur est appliqué à une List ou à un tableau, la page tente de convertir la valeur de droite en index, et accède au résultat en utilisant les méthodes get( ) adaptée à la circonstance :

```
unObjet.get(nom) ; // unObjet est une List --- appel de la méthode get(int)
Array.get(unObjet, nom) ; // unObjet est un tableau --- appel de la méthode get(Object, int)
```

Conversions et valeurs par défaut

EL peut effectuer un certain nombre de conversions de façon implicite. Parmi les conversions, il y a bein entendu les conversions déjà supportées par Java comme les conversions d'une valeur entière en valeur double, mais aussi la conversion des types objet encapsulant des valeurs scalaires vers ces valeurs scalaires.

Ansi, si x est une référence sur une instance de java.lang.Integer, l'expression :

```
${x+3} // est acceptée par EL. Son évaluation extrait la valeur entière encapsulée par l'objet référencé par x, et ajoute 3 à cette valeur.
```

De plus, les conversions automatiques sont faites aussi sur les chaînes de caractères. Ainsi :

```
${param.x+2} converti la chaine param.x en valeur entière.
```

Au passage, remarquons que le symbole + dénote toujours l'addition. En effet, param.x est une chaîne de caractère puisque c'est la valeur du paramètre x de la requête. En java pur, le symbole + serait interprété, dans ce cas là, comme la concaténation de cette chaîne avec la chaine "2". En EL, au contraire, le symbole + est ici compris comme l'addition de la conversion de param.x en entier de 2. Cette expression invoque donc implicitement la méthode Integer.parseInt().

Gestion des exceptions à l'intérieur d'une expression

Une expression EL fournit toujours une valeur, même lorque une exception raisonnable survient. Il s'agit, à ce moment là, d'une valeur par défaut. Par exemple, si nous essayons d'accéder à une propriété d'un objet via une référence nulle, EL traite l'exception java.lang.NullPointerException et renvoie la valeur null. A son tour, la valeur null est remplacée par la chaîne vide si le contexte attend une chaîne, 0 si le contexte attend une valeur numérique.

Ainsi, l'expression ${ param.x + 15 } vaut 15 si la requête http ne possède pas de paramètres x.

De même, l'expression ${ unTableau[5] } renvoie null si la variable unTableau vaut null ou si elle référencie un tableau ou une liste ayant moins de six éléments.

Les objets implicites

Les expressions EL ont accès à des objets implicites. La plupart sont les mêmes que ceux disponibles dans les scriptlets et les expressions JSP. Grâce à ces objets, les expressions EL permettent de réaliser la plupart des tâches qui peuvent être effectuées par ces dernières. Les objets implicites sont les suivants :

1. *pageContext - L'objet de type javax.servlet.jsp.PageContext correspond à la page. Il peut être employé pour accéder ensuite aux objets implicites utilisés par les pages JSP : request, response, session, out, etc. Par exemple, ${pageContext.request} renvoie l'objet request de la page.*
2. *pageScope - Une Map permettant d'accéder aux objets dont la portée est page. Ainsi, si un objet est placé dans la page avec le nom monObjet, une expression EL est en mesure d'y accéder grâce à la syntaxe ${pageScope.monObjet}. Pour accéder à une des propriétés de l'objet, il suffit d'utiliser la syntaxe ${pageScope.monObjet.nomPropriété}.*
3. *requestScope - Une Map permettant d'accéder aux objets dont la portée est request. Cet objet permet d'accéder aux attributs de l'objet request.*
4. *sessionScope - Une Map permettant d'accéder aux objets dont la portée est session. Cet objet permet d'accéder aux attributs de l'objet session.*
5. *applicationScope - Une Map permettant d'accéder aux objets dont la portée est application. Cet objet permet d'accéder aux attributs de l'objet application.*
6. *param - Une Map contenant les valeurs des paramètres de la requête sous forme de paires formées d'un nom de paramètre et de la représentation de sa valeur sous forme d'une chaîne de caractères. Souvenez-vous que l'objet request contient les données envoyées par le client. La méthode getParameter(String) retourne la valeur du paramètre dont le nom est indiqué. L'expression ${param.nom} est équivalente à request.getParameter(nom). (Notez que nom n'est pas la chaîne 'nom', mais le nom du paramètre).*
7. *paramValues - Une Map mettant en relation le nom de chaque paramètre avec un tableau de String représentant ses valeurs (obtenues en appelant la méthode ServletRequest.getParameterValues(String nom)). La différence avec l'objet précédent est que le résultat est un tableau de valeurs et non une valeur unique. Par exemple, ${paramValues.nom} est équivalent à request.getParameterValues(nom).*
8. *header - Une Map contenant les en-têtes HTTP et leurs valeurs (obtenues grâce à la méthode ServletRequest.getHeader(String nom)). Les requêtes contiennent toujours un certain nombre d'en-têtes : type de contenu, longueur, cookies, URL référente, etc. L'expression ${header.nom} est équivalente à request.getHeader(nom).*
9. *headerValues - Une Map mettant en relation le nom de chaque en-tête avec un tableau de String représentant ses valeurs (obtenues grâce à la méthode ServletRequest.getHeaders(String nom)). Cet objet est semblable à l'objet implicite header. L'expression ${headerValues.nom} est équivalente à request.getHeaderValues(nom).*
10. *cookie - Une Map associant les objets Cookie à leurs noms. Un client peut envoyer un ou plusieurs cookies au serveur avec sa requête. L'expression ${cookie.nom.value} retourne la valeur du premier cookie portant le nom indiqué. Si la requête est susceptible de contenir plusieurs cookies de même nom, il faut utiliser la méthode ${headerValues.nom}.*
11. *initParam - Une Map associant les noms des paramètres d'initialisation et leur valeurs (obtenues grâce à la méthode ServletRequest.getInitParameter(String nom)). Pour accéder à un paramètre d'initialisation, utilisez ${initParam.nom}.*

Contexte de page

Le contexte de page est une référence sur l'objet décrivant le contexte d'exécution de la page. Il est représenté par l'objet implicite pageContext.

ATTENTION, les variables déclarées dans les scripts de déclaration <%= variable %> ne sont pas accessibles directement par les expressions. Il faut placer les variables qui seront ensuite exploitées par les expressions EL au moyen de la méthode setAttribute() de l'objet implicite pageContext.

La méthode setAttribute() attend deux paramètres : Une chaîne de caractères qui correspond au nom donné à l'attribut suivi de la valeur à stocker (doit être également une chaîne de caractères). En fait, cet attribut reprend la philosophie du couple nom/valeur comme pour un paramètre de requête. C'est pour cette raison que nous ne devons avoir que des chaînes de caractères. Une fois que cette opération est réalisée, nous pouvons récupérer la valeur stockée à l'aide de l'expression EL en spécifiant juste le nom de l'attribut.

```
<% String message = "Mon message";          /
        pageContext.setAttribute("texte", message); %>// Place message dans le contexte de page : pageContext
...
${ texte}   //utilisation  de l'attribut texte à l'aide de l'expression EL. ATTENTION, c'est le nom de l'attribut qu'il faut prendre et pas la variable message.
```

Quelques exemples qui exploitent ces objets implicites.

```
${pageContext.request.remoteHost} // Récupérer l'adresse IP du client :
```

Dans le code suivant, le bean a la portée page et possède une propriété prénom :

```
<jsp:useBean id = "utilisateur"  scope = "page" class = "Personne">
        <jsp:setProperty name = "utilisateur" property = "prénom" />
</jsp:useBean>
...
${pageScope.utilisateur.prénom} // Renvoie la propriété prénom du bean utilisateur.
```

Si vous avez placé un objet dans la session, vous pouvez y accéder de la manière suivante :

```
<% session.put("adresse", "12, rue de Java."); %>
${sessionScope.adresse} // renvoie '12, rue de Java'
<%= session.get("adresse") %>  // expression JSP équivalente
```

Nous pouvons obtenir la valeur d'un champ d'en-tête de la requête par l'objet implicite header :

```
${header["accept-encoding"]}
```

L'objet implicite initParam désigne la table des paramètres d'initialisation de l'application Web :

```
${initParam.couleurFond}
```

Paramètres de la requête

EL désigne donc les paramètres de la requête au moyen de l'objet implicite param. Cet objet est une collection (Map) dont les éléments peuvent être désignés via les opérateurs [ ] et . :

expression	valeur
${empty param}	vaut true si aucun paramètre
${empty param["x"]}	true si le paramètre x est absent, false sinon.
${empty param.x}	true si le paramètre x est absent, false sinon.
${param["x"]}	valeur du paramètre x s'il est présent, chaîne vide sinon.
${param.x}	valeur du paramètre x s'il est présent, chaîne vide sinon.
${param[1]}	valeur du paramètre x s'il est présent, chaîne vide sinon.

Les JavaBeans

Les expressions EL offre un accès direct à tous les objets placés en attributs dans l'un des quatre scopes standard :

```
<jsp:useBean id="compteur" class="Compteur" scope="application" />
...
${compteur.valeur+1}
```

Dans cet exemple, EL résoud l'identificateur compteur en retrouvant le bean placé en attribut de l'application puis résoud l'identificateur valeur comme étant une propriété de ce bean. Le message getValeur() est alors envoyé au bean. Au besoin, le résultat de celle-ci est utilisé par le contexte (la page).

Attention, EL ne peut pas accéder aux variables Java définies dans la page JSP, EL n'accède qu'aux attributs des quatre scopes standard.
.

Utilisation des expression EL

Les expressions EL peuvent être utilisées comme attributs des actions standards et personnalisées. Elles peuvent aussi être palacées ailleurs dans le texte de la page, c'est-à-dire dans le code HTML. Le code suivant montre l'utilisation d'une expression EL comme attribut d'une action standard <jsp:forward> :

```
<jsp:forward page=" ${param.nextPage}" />
```

Dans cet exemple, l'action <jsp:forward> transmet la requête à l'URL spécifiée par le paramètre de la requête nextPage. Si le paramètre n'existe pas ou si la valeur n'est pas une URL valide, la page produit une erreur.

Modification de notre application Web messagerie afin de tenir compte des expressions EL

Nous allons juste montrer le changement obtenu avec l'utilisation des expressions EL sur deux pages JSP : le fragment de page <navigation.jspf> et la page <validerutilisateur.jsp> :

navigation.jspf

```
1  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
2
3  <%@ page errorPage = "/WEB-INF/erreur.jsp" import="bd.*" %>
4
5  <%! int idPersonne = 1; %>
6
7  <%
8      String identité = "A tout le monde";
9      Personne opérateur = (Personne) session.getAttribute("utilisateur");
10     if (opérateur != null) {
11         idPersonne = opérateur.identificateur();
12         identité = opérateur.getPrénom()+" "+opérateur.getNom();
13     }
14     pageContext.setAttribute("personne", identité);
15 %>
16
17 <html>
18   <head><title>Messages</title></head>
19   <body bgcolor="#${initParam.couleurFond}">
20     <font face="Arial">
21         <h2 align="center">Messages</h2>
22         <hr>
23         <table bgcolor="1" cellpadding="3" cellspacing="2" width="90%" align="center">
24             <tr bgcolor="#FF9900">
25                 <th align="left"><a href="bienvenue">Sujets</th>
26                 <th align="right">
27                     <% if (idPersonne == 1) { %>
28                         <a href="utilisateur?authentification=personnel">Identification</a>
29                         <a href="utilisateur?authentification=nouveau">Inscription</a>
30                     <% }
31                        else { %>
32                         <a href="#">Nouveau</a>
33                         <a href="#">Modifier</a>
34                         <a href="#">Enlever</a>
35                     <% } %>
36                 </th>
37             </tr>
38             <tr>
39                 <th align="left">${personne}</th>
40                 <th align="right">${pageContext.request.remoteHost}</th>
41             </tr>
42         </table>
43
```

Ligne 8 :

Cette fois-ci la chaîne **identité** n'est plus déclarée dans un script de type déclaration mais dans une scriptlet classique. Toutefois, **identité** va être utilisée ultérieurement dans la page, mais cette fois-ci par une expression EL. Hors, cette dernière ne peut pas accéder directement aux scripts de type déclaration.

Ligne 14 :

Pour pouvoir récupérer l'identité de la personne, il faut la placer dans le contexte de page afin qu'elle soit accessible par les expressions EL. Ici, le nom de l'attribut est **personne**, et c'est lui qu'il faudra solliciter pour récupérer effectivement la valeur de l'identité de la personne.

Lignes 19 :

Récupération du paramètre d'intitialisation **couleurFond** de l'application Web.

Lignes 39 :

Récupération de l'attribut **personne** préalablement stockée dans le contexte de page.

Ligne 40 :

Récupération de l'adresse IP de l'ordinateur client.

validerutilisateur.jsp

```
 1 <%@ include file = "/WEB-INF/jspf/navigation.jspf" %>
 2
 3 <h3 align="center">Confirmation de votre demande d"inscription</h3>
 4
 5 <jsp:useBean id="utilisateur" class="bd.Personne" scope="session">
 6     <jsp:setProperty name="utilisateur" property="*" />
 7
 8     <p><table border="1" cellpadding="3" cellspacing="2" width="90%" align="center">
 9         <tr>
10             <td bgcolor="#FF9900" width="100"><b>Nom</b></td>
11             <td>${utilisateur.nom}</td>
12         </tr>
13         <tr>
14             <td bgcolor="#FF9900" width="100"><b>Prénom</b></td>
15             <td>${utilisateur.prénom}</td>
16         </tr>
17         <tr>
18             <td bgcolor="#FF9900" width="100"><b>Mot de passe</b></td>
19             <td>${utilisateur.motDePasse}</td>
20         </tr>
21     </table></p>
22 </jsp:useBean>
23
24 <h3 align="center">
25 <% if (!utilisateur.enregistrer()) {  %>
26      <font color="red">ATTENTION : Utilisateur déja enregistré</font>
27 <%
28         session.removeAttribute("utilisateur");
29     } else {
30 %>
31      <font color="green">Nouvel utilisateur enregistré</font>
32 <% } %>
33 </h3>
34
35 <%@ include file = "/WEB-INF/jspf/pieds.jspf" %>
```

Ligne 11, 15, 19 :

Récupération des propriétés **nom**, **prénom** et **motDePasse** du bean **utilisateur**.

Les bibliothèques de balises personnalisées

A plusieurs reprises, aux cours des études précédentes, nous avons évoqué la possibilité de débarrasser les pages JSP de tout code Java, de façon à séparer autant que possible l'affichage de la logique métier. En fait, il ne s'agit pas vraiment de supprimer le code Java, mais de le masquer de façon qu'il soit invisible pour les développeurs des pages. Par exemple, dans notre application Web, nous avons employé certaines actions standards définies par la spécification. Les actions standards sont des actions qui doivent être implémentées par tous les conteneurs JSP.

Une action standard est notée sous la forme d'éléments délimités par des balises XML. Voici l'élément correspondant à l'action useBean. Il contient lui-même une action setProperty :

```
<jsp:useBean id="utilisateur" class="bd.Personne">
        <jsp:setProperty name="utilisateur" property="*" />
</jsp:useBean>
```

Une balise commence par le préfixe d'espace de noms jsp (les espaces de nom sont l'équivalent XML des packages Java). Le préfixe est suivi du nom de l'action. Les actions standards peuvent avoir un certain nombre d'attributs, et certianes ont un contenu, placé entre la balise d'ouverture et la balise de fermeture. Le contenu peut inclure d'autres balises, comme dans l'exemple ci-dessus, ou du texte ou du code HTML. Ces balises utilisent la syntaxe XML.

Le traducteur JSP voit les balises sous un angle un peu différent. De son point de vue, il s'agit de marqueurs devant être remplacés par du code Java implémentant la fonctionnalité de la balise. Ainsi, le code Java n'est pas retiré des pages, il est simplement encapsulé dans les balises. Lorsque le traducteur de JSP rencontre la première balise de l'exemple précédent, il la remplace par :

```
utilisateur = (bd.Personne) java.beans.Beans.instanciate(this.getClass().getClassLoader(), "bd.Personne") ;
```

Si les actions disponibles étaient les actions standards, nous ne pourrions pas nous passer de code Java dans les pages JSP. Heureusement, la spécification prévoit un moyen pour les développeurs de créer de nouvelles actions, appelées actions personnalisées. Ces actions sont déployées à l'aide des bibliothèques de balises personnalisées (custom tag libraries). Le mécanisme consistant à définir, implémenter, déployer et exécuter les actions personnalisées est appelé extension de balises (tag extension). A l'aide des actions standards et des actions personnalisées, un concepteur de pages Web peut créer des pages dynamiques sans avoir la moindre connaissance de Java.

Différentes phases dans la création de balises personnalisées

La création de balises Java passe par les étape suivantes :

1. *Ecriture des classes Java implémentant les balises à créer,*

2. *Création des fichiers de description des différentes balises (**bibliothèques de balises personnalisées**),*
3. *Modification du descripteur de déploiement <web.xml> pour que l'application Web connaisse ces balises aux démarrages.*
4. *Utilisation de ces balises dans les pages JSP.*

Dans la suite de ce chapitre, nous allons commencer par la fin, c'est-à-dire par l'utilisation des balises qui représentent les actions personnalisées.

Les actions personnalisées

Le terme action personnalisée (et action standard) fait généralement référence à une balise dans une page JSP. Les actions personnalisées peuvent être employée dans les pages JSP comme des balises ordinaires. Elles sont identifiées par un préfixe et un nom :

```
<préfixe:nom />
```

Le préfixe permet d'éviter les conflits de noms entre les balises de différentes bibliothèques. Il est choisi par le développeur de pages, bien que le développeur de la bibliothèque de balises puisse suggérer un nom. Le préfixe est suivi du nom de l'action, qui est choisi par le développeur de la bibliothèque.

Les actions personnalisées peuvent être vides :

```
<x:actionPersonnalisée /> // Balise d'ouverture et de fermeture combinées en une seule
<x:actionPersonnalisée></x:actionPersonnalisée> // balises séparées
```

Elles peuvent avoir un corps :

```
<x:actionPersonnalisée>
        Corps de l'action
</x:actionPersonnalisée>
```

Le code Java implémentant la balise peut évaluer le corps de l'action, ou au contraire l'ignorer. Les actions peuvent être imbriquées, comme pour les beans :

```
<jsp:useBean id="utilisateur" class="bd.Personne">
        <jsp:setProperty name="utilisateur" property="*" />
</jsp:useBean>
```

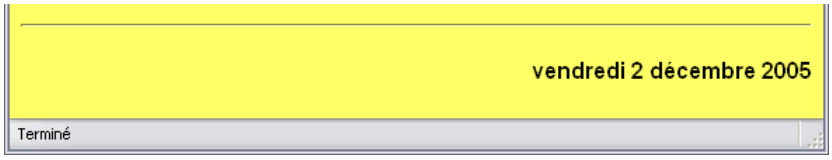
Comme le montre les exemples de <jsp:useBean> et <jsp:setProperty>, les actions peuvent avoir des attributs qui spécifient les détails de leur comportement. Elles ont accès aux objets implicites (request, response, session, etc.) et peuvent les utiliser pour modifier la réponse renvoyée au client. Les actions personnalisées peuvent créer des objets qui pourront ensuite être manipulés par d'autres actions ou des scriptlets.

Le comportement d'une action est défini lors de l'exécution par une instance d'une classe Java. Cette classe est appelée gestionnaire de balises.
.

Afin de bien maîtriser tous ces concepts, nous allons nous servir de l'applications Web Messagerie à l'intérieur de laquelle nous allons définir un certain nombre de balises personnalisées qui va permettre d'alléger considérablement le code des pages JSP.

Balise personnalisée <:date> pour notre application Web Messagerie

A titre d'exemple, nous allons créer plusieurs balises personnalisées qui vont nous permettre de simplifier considérablement l'écriture des pages JSP du site de la messagerie. La première balise à fabriquer consiste à afficher la date du jour avec le format français adapté, tel que nous l'avons déjà mis en oeuvre dans le pieds de page <pieds.jspf> :



pieds.jspf

```
<%@page import="java.util.Date, java.text.DateFormat" %>

<%!
    DateFormat formatDate = DateFormat.getDateInstance(DateFormat.FULL);
 %>
        <br><hr>
        <h4 align="right"><%= formatDate.format(new Date()) %></h4>
    </font>
  </body>
</html>
```

Vous remarquez que sur cette simple petite page JSP (fragment), nous avons pas mal de code Java intégré. L'investissement qui consiste à créer une nouvelle balise personnalisée vaut le coup ici. Surtout que ce type de balise peut être utilisé pour bien d'autres applications Web. Ainsi, il serait judicieux de nommer cette balise <:date> ou éventuellement <util:date> si nous utilisons le préfixe correspondant à l'espace de nom (le " : " est de toute façon nécessaire). Voici du coup le changement obtenu avec l'utilisation d'une telle balise :

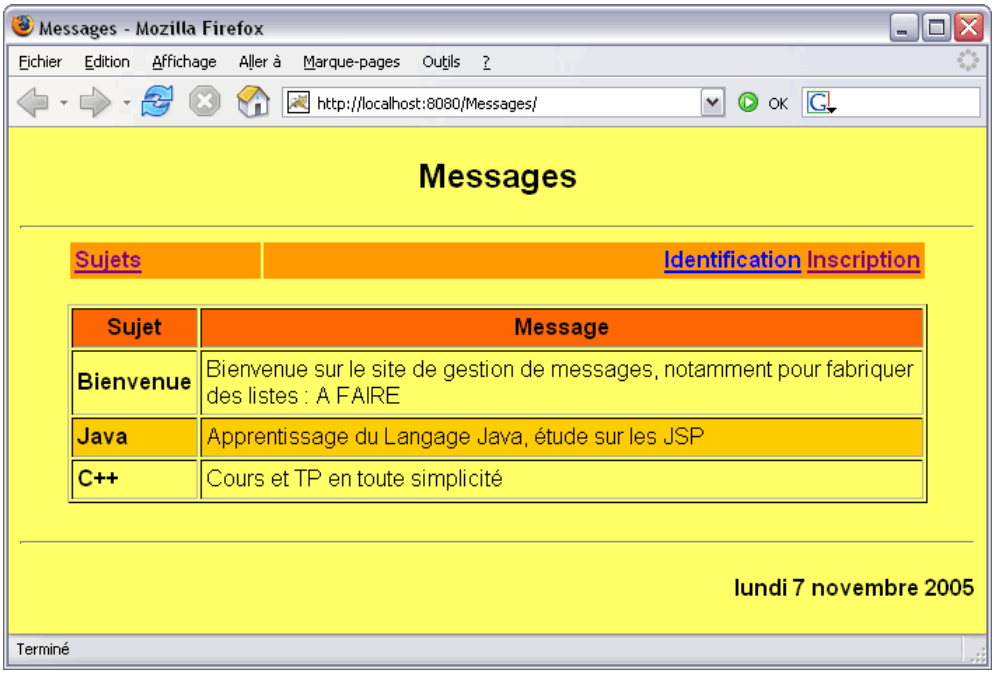
pieds.jspf

```
        <br><hr>
        <h4 align="right"><:date></h4>
    </font>
  </body>
</html>
```

Le changement est ici considérable. La page gagne largement en clarté avec en plus, très peu de lignes de code. Même pour un développeur Java, sans parler du webmaster, il est préférable de travailler de cette façon. Il est en effet fréquent de ne plus se souvenir des classes et des méthodes à utiliser pour réaliser une opération particulière, comme ici, le formatage d'une date. Une simple balise nous permet de réaliser l'opération souhaitée.

Balises personnalisées permettant d'afficher la liste des messages de notre application Web

Passons maintenant à la deuxième page que je désire modifier. Il s'agit de la page <bienvenue.jsp> qui propose entre autre le listing des messages associés à la personne. Je rappelle, ci-dessous, l'affichage de cette page :



bienvenue.jsp

```
 2 <%@ include file = "/WEB-INF/jspf/navigation.jspf" %>
 3
 4 <font face="Arial">
 5
 6 <p><table border="1" cellpadding="3" cellspacing="2" width="90%" align="center">
 7     <tr bgcolor="#FF6600">
 8         <th>Sujet</th>
 9         <th>Message</th>
10     </tr>
11     <%
12         ListeMessages listeMessages = new ListeMessages(idPersonne);
13         int ligne = 0;
14         while (listeMessages.suivant()) {
15     %>
16     <tr bgcolor="<%= ligne++ % 2 == 0 ? "#FFFF66" : "#FFCC00" %>">
17         <td><b><%= listeMessages.sujet() %></b></td>
18         <td><%= listeMessages.texte() %></td>
19     </tr>
20     <%
21         }
22         listeMessages.arrêt();
23     %>
24 </table></p>
25 </font>
26
27 <%@ include file = "/WEB-INF/jspf/pieds.jspf" %>
```

Le but ici est de créer des actions imbriquées afin de représenter un tableau avec toutes les colonnes désirées de n'importe quelle table issue de n'importe quelle base de données. Ainsi, les balises personnalisées que nous allons mettre en oeuvre pourront être déployées sur d'autres applications Web. Le nom de l'utilisateur de la base de données ainsi que son mot de passe ne doivent pas apparaître, ils seront collectés à partir des paramètres d'initialisation de l'application Web, qu'il faut définir dans le descripteur de déploiement. Voici, ce que nous pourrions obtenir en prenant en compte toutes ces considérations :

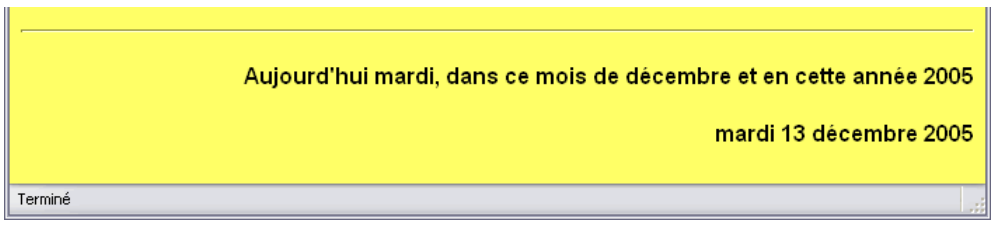
bienvenue.jsp

```
 2 <%@ include file = "/WEB-INF/jspf/navigation.jspf" %>
 3
 4 <font face="Arial">
 5
 6 <p><util:tableau bd="messagerie" table="message" identificateur="idPersonne" valeur="${idPersonne}" couleur="#FF6600"
largeur="90%">
 7    <util:colonne titre="sujet">Sujet</util:colonne>
 8    <util:colonne titre="texte">Message correspondant</util:colonne>
 9 </util:tableau></p>
25 </font>
26
27 <%@ include file = "/WEB-INF/jspf/pieds.jspf" %>
```

Encore une fois, le changement est considérable. Ici aussi, la page gagne largement en clarté avec très très peu de lignes de code. La première balise <util:tableau> permet de régler le tableau dans son entier afin qu'il soit en correspondance directe avec la base de données en précisant tous les critères nécessaires afin de collecter toutes les informations à afficher. L'apparence est également prise en compte en spécifiant la taille désirée ainsi que la couleur de fond de la première ligne du tableau correspondant aux intitulés. Les intitulés sont spécifiés par l'intermédiaire des balises <util:colonne>. L'attribut "titre" de la balise <util:colonne> indique la colonne de la table de la base de données à prendre en compte.

Changement de comportement de la balise personnalisée <:date>

Nous pouvons proposer une autre alternative dans l'utilisation de la balise <:date>. Cette fois-ci en effet, l'action personnalisée consistera à créer des attributs de page afin de laisser au webmaster le libre choix de la représentation de la date. En voici d'ailleurs un exemple ci-dessous :



pieds.jspf

```
<%@taglib uri="/util" prefix="u" %> <u:date />

        <br><hr>
        <h4 align="right">
            Aujourd"hui ${jourSemaine}, dans ce mois de ${mois} et en cette année ${année}
        </h4>
        <h4 align="right">${date}</h4>
     </font>
   </body>
</html>

```

Le code reste clair. Nous utilisons juste des expressions EL. La particularité cette fois-ci, c'est que lorsque nous plaçons la balise <u:date />, aucun affichage n'est proposé. Cette balise s'occupe essentiellement de créer des attributs de pages que nous pouvons utiliser par la suite où bon nous semble (sur toute la page). Nous utilisons dans cet exemple quatre attributs : jourSemaine qui retourne 'mardi', mois qui retourne 'décembre', année qui retourne '2005' et date qui retourne la date complète.

Contrôle du type d'authentification <:authentification> de notre application Web

Les balises que nous venons de voir pour l'instant peuvent s'appliquer quel que soit l'application Web. Elles sont suffisamment génériques pour cela. Il est toute fois possible de créer des balises qui ne seront utiles que pour cette application Web. Le but est le même, il s'agit de faire en sorte que les pages JSP soient faciles à construire, sans trop de lignes de code, pour que par la suite elles soient plus faciles à lire. De toute façon, l'investissement vaut le coup. Il faut toujours penser à la suite. Plus les pages sont concises, plus elles sont faciles à maintenir ou à mettre à jour. Cela permet de bien partager les compétences.

Nous allons donc mettre en oeuvre une balise spécifique qui ne fonctionnera que pour cette application Web. Souvenez-vous que la première page de notre application nous dirige en fonction de l'authentification réalisée par l'opérateur. Si nous souhaitons nous inscrire ou consulter nos propres messages, nous passons systématiquement par la page <utilisateur.jsp>. Du coup, il est nécessaire de connaître quel est le type d'authentification afin d'avoir, d'une part la bonne visualisation en conséquence, et d'autre part de connaitre la prochaine page à afficher. Voici la page JSP que nous possédons à l'heure actuelle :

utilisateur.jsp

```
<%@ page errorPage = "/WEB-INF/erreur.jsp" import = "bd.*" %>
<%@ include file = "/WEB-INF/jspf/navigation.jspf" %>

<%! boolean nouveau; %>
<% nouveau = request.getParameter("authentification").equals("nouveau"); %>

<h3 align="center">
    <font color="green"><%= (nouveau ? "Demande d"inscription" : "Vos références") %></font>
</h3>

<form action="<%= (nouveau ? "validerutilisateur.jsp": "controleidentite.jsp") %>" method="post">
    <p><table border="1" cellpadding="3" cellspacing="2" width="90%" align="center">
        <tr>
            <td bgcolor="#FF9900" width="100"><b>Nom</b></td>
            <td><input type="text" name="nom"></td>
        </tr>
        <tr>
            <td bgcolor="#FF9900" width="100"><b>Prénom</b></td>
            <td><input type="text" name="prénom"></td>
        </tr>
        <tr>
            <td bgcolor="#FF9900" width="100"><b>Mot de passe</b></td>
            <td><input type="password" name="motDePasse"></td>
        </tr>
    </table></p>
    <p align="center"><input type="submit" value="Valider"></p>
</form>

<%@ include file = "/WEB-INF/jspf/pieds.jspf" %>
```

Dans cette page, nous trouvons des déclarations ainsi que des expressions. Il peut être alors judicieux de proposer une balise <:authentification> qui permet d'enlever tout ce code Java et donc de rendre la page plus claire.

utilisateur.jsp

```
<%@ include file = "/WEB-INF/jspf/navigation.jspf" %>
<%@taglib uri="/personnel" prefix="personnel" %>

<h3 align="center">
    <font color="green"><personnel:authentification /></font>
</h3>

<form action="${prochainePage}" method="post">
  <p><table border="1" cellpadding="3" cellspacing="2" width="90%" align="center">
    <tr>
       <td bgcolor="#FF9900" width="100"><b>Nom</b></td>
       <td><input type="text" name="nom"></td>
    </tr>
    <tr>
       <td bgcolor="#FF9900" width="100"><b>Prénom</b></td>
       <td><input type="text" name="prénom"></td>
    </tr>
    <tr>
```

```
        <td bgcolor="#FF9900" width="100"><b>Mot de passe</b></td>
        <td><input type="password" name="motDePasse"></td>
    </tr>
  </table></p>
  <p align="center"><input type="submit" value="Valider"></p>
</form>

<%@ include file = "/WEB-INF/jspf/pieds.jspf" %>
```

La balise <personnel:authentification /> s'occupe de récupérer l'authentification délivrée par la page <authentifier.jsp>. L'en-tête de page s'affiche en conséquence. Cette balise fabrique également l'attribut de page 'prochainePage' qui est utilsé ensuite par la balise <form> et qui représente la prochaine page à activer à la suite de la validation du formulaire par l'opérateur. Bien que cela ne soit pas du tout obligatoire, j'ai utilisé un espace de nom différent pour faire la séparation entre les balises pouvant être déployées sur d'autres applications Web et celles qui sont spécifiques à cette application Web.

Vérification de l'identité de l'utilisateur <:verifier>

La dernière balise que nous allons mettre en oeuvre est juste là à titre pédagogique. Son intérêt peut être discutable. Toutefois, cette balise, <:verifier />, sera capable de récupérer les valeurs d'un bean, d'être en relation avec la session, et enfin de rediriger la requête vers la page d'accueil <bienvenue.jsp>. Nous allons donc changer le code de la page <controleidentite.jsp> dont voici les quelques lignes :

controleidentite.jsp

```
<%@ page errorPage = "/WEB-INF/erreur.jsp" import="bd.*" %>

<jsp:useBean id="utilisateur" class="bd.Personne" scope="session">
    <jsp:setProperty name="utilisateur" property="*" />
</jsp:useBean>

<%  if (!utilisateur.authentifier()) {
        session.removeAttribute("utilisateur");
    }
%>

<jsp:forward page="bienvenue.jsp" />

```

Ainsi, toute la partie contrôle d'identité, suppression de l'attribut utilisateur dans la session, et redirection au moyen de l'action <jsp:forward /> va être résolu au moyen d'une seule balise <:verifier /> dont voici l'écriture ci-dessous :

controleidentite.jsp

```
<%@ page errorPage = "/WEB-INF/erreur.jsp" import="bd.*" %>
<%@taglib uri="/personnel" prefix="personnel" %>

<jsp:useBean id="utilisateur" class="bd.Personne" scope="session">
    <jsp:setProperty name="utilisateur" property="*" />
</jsp:useBean>

<personnel:verifier />
```

Au travers de ces quelques exemples, je pense que nous aurons fait le tour des cas d'utilisation possibles. En tout cas, vous en avez le principe.
.

Le comportement d'une action est défini lors de l'exécution par une instance d'une classe Java. Cette classe est appelée gestionnaire de balises.
.

Les gestionnaires de balises

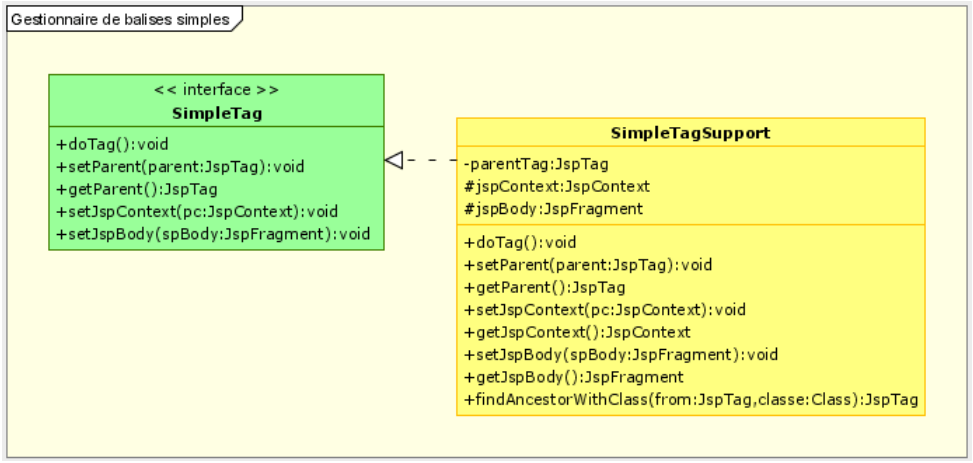
Le gestionnaire de balises est la classe Java chargée d'implémenter le comportement d'une action. cette classe doit respecter les spécifications d'un JavaBean et doit implémenter une des interfaces d'extension de balises. Plusieurs de ces interfaces sont disponibles (suivant les spécifications) :

Spécification JSP	Interfaces JSP	Explications
JSP 1.1	Tag, BodyTag	L'interface Tag est employée pour implémenter une action simple, sans itération et sans évaluation du corps. BodyTag permet de traiter le cas où le corps doit être évalué (et non simplement affiché).
JSP 1.2	IterationTag	La spécification JSP 1.2 a introduit l'interface IterationTag qui permet de gérer les itérations plus facilement que BodyTag. Ces trois interfaces (Tag, BodyTag et IterationTag) sont appelées gestionnaires de balises classiques.
JSP 2.0	SimpleTag, JspFragmentTag	La spécification JSP 2.0 ajoute l'interface SimpleTag, qui facilite le traitement des balises, et JspFragmentTag, qui permet d'encapsuler le contenu du corps de l'action dans un objet. Ces deux interfaces sont appelées gestionnaires de balises simples.

Les gestionnaires de balises simples sont dits simples car ils simplifient le développement des gestionnaires de balises. Ils ne sont pas pour autant moins complets que les gestionnaires classiques en ce qui concerne les itérations et le traitement du corps des actions. C'est donc probablement ceux que vous utiliserez le plus fréquemment.

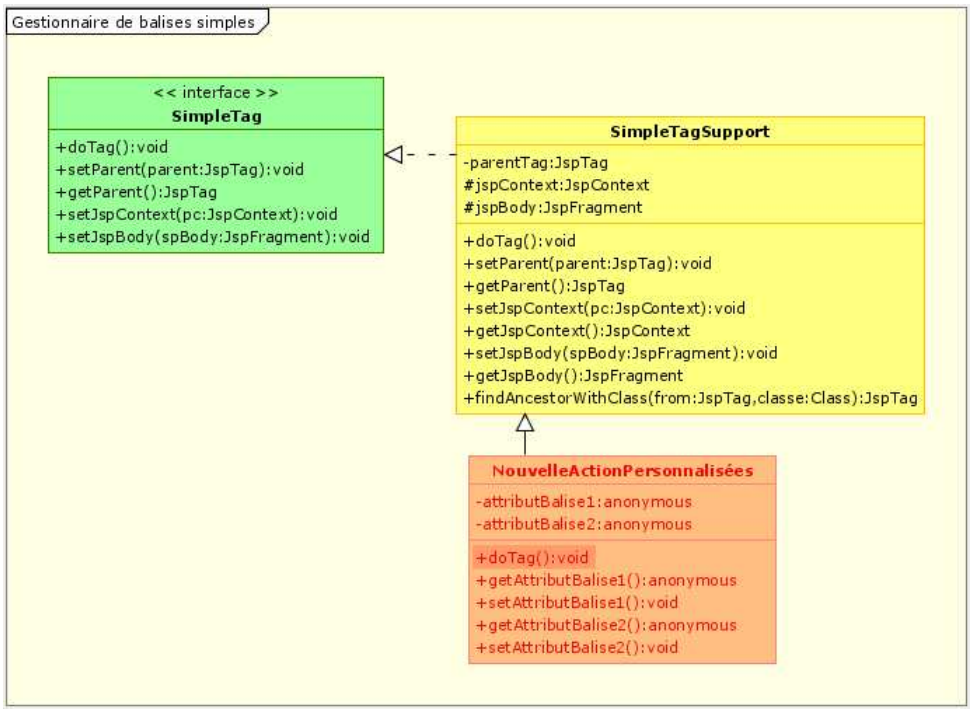
Gestionnaires de balises simples

Le mécanisme d'extension de balises de la JSP 1.2 était très puissant, mais relativement complexe. JSP 2.0 ajoute l'interface SimpleTag ainsi qu'une classe de base, SimpleTagSupport, implémentant cette interface.



L'interface SimpleTag et la classe SimpleTagSupport permettent d'implémenter tous les gestionnaires de balises JSP 2.0, avec ou sans itération et évaluation du corps. Pour créer une action personnalisée, il suffit alors de créer une classe étendant la classe de base SimpleTagSupport en redéfinissant les méthodes nécessaires pour produire le comportement souhaité, y compris les itérations et l'évaluation du corps. Le plus souvent, la seule méthode nécessaire est doTag(). Cette méthode gère l'intégralité du comportement de l'action.

Les gestionnaires classiques ont besoin de trois méthodes pour faire la même chose.
.



Lorsque le traducteur rencontre une balise dans une page JSP, il génère le code nécessaire pour :

1. *créer une instance du gestionnaire de balises ;*
2. *appeler les méthodes **setJspContext()** et **setParent()** ;*
3. *initialiser les attributs (propriétés) du gestionnaire de balises ;*
4. *créer un objet **JspFragment** et appeler **setJspBody()**.*

Pour que le code de l'implémentation de la page soit en mesure de créer l'instance du gestionnaire et d'initialiser ses attributs (propriétés), il doit respecter les conventions JavaBean :

1. *Il doit posséder un constructeur sans argument (constructeur par défaut).*
2. *posséder des attributs privées qui vont jouer le rôle de propriétés.*
3. *Ses propriétés doivent être exposées publiquement au moyen d'accesseurs, c'est-à-dire de méthodes **getXxx()** (pour lire sa valeur) et **setXxx()** (pour modifier une propriété.*

Une propriété est composée de trois éléments : d'abord un attribut privé suivi des deux accesseurs. Le nom de chaque accesseur, appelés communément getter et setter est construit avec get ou set suivi du nom de la propriété (attribut) avec la première lettre transformée en majuscule.

IMPORTANT, ici les propriétés du gestionnaire de balises correspondent aux attributs de la balise personnalisée.
.

De cette façon, les conteneurs de JSP disposent d'une procédure standard afin de créer et d'initialiser les instances de gestionnaires des actions à partir des attributs présents dans les balises. Chacun des attributs de balise doit correspondre à une propriété du gestionnaire pouvant être initialisée au moyen d'une méthode setXxx().

Une fois la classe du gestionnaire créée et initialisée, la page appelle sa méthode doTag(). Cette méthode est appelée juste une fois par balise ; si le corps de l'action doit être évalué, cette évaluation est confiée à un objet JspFragment qui lui est passé grâce à la méthode setJspBody().

Nous pouvons récupérer les informations de la page JSP contenant la balise personnalisée au moyen de la méthode getJspContext() de la classe de base SimpleTagSupport. C'est au travers de cette méthode que vous allez établir les différentes relations entre la page JSP et la classe implémentant votre balise personnalisée.

JspFragment

Comme SimpleTag, JspFragment est une interface, mais son implémentation est laissée au conteneur de JSP. En tant que développeur, vous avez seulement besoin de savoir comment appeler les méthodes d'un fragment JSP pour évaluer son contenu. Si votre gestionnaire SimpleTag doit évaluer le contenu d'une balise, il doit appeler la méthode invoke() de JspFragment :

```
public void invoke(java.io.Writer out)
```

Comme vous pouvez le voir, invoke() prend pour seul argument un Writer. Si cet argument est null, le fragment envoie les données dans le Writer de la réponse au client. Dans le cas contraire, le Writer fourni est utilisé. Les JspFragment peuvent contenir du texte, des actions JSP et des expressions EL mais pas de scriptlets, ni d'expressions JSP. Les variables employées dans les expressions EL sont initialisées grâce aux attributs du contexte, comme nous le verrons dans le prochain exemple.

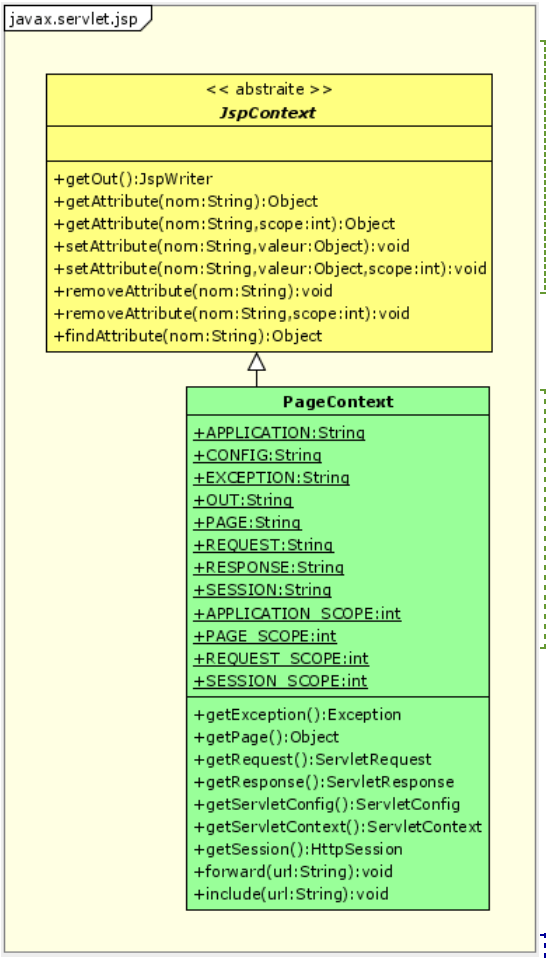
Méthodes à connaître pour gérer l'ensemble des situations possibles

Les méthodes de la classe de base SimpleTagSupport du gestionnaire de balises simples :

1. *Object getParent() : Cette méthode permet de connaître la balise parente de la balise sur laquelle nous travaillons (retourne l'instance de la classe englobante). Ainsi, grâce à cette méthode, il est facile de gérer un système de balises imbriquées.*
2. *JspContext getJspContext() : récupère tout le contexte de la page JSP qui utilise cette balise. Grâce à cette méthode, il est alors possible de travailler avec des objets équivalents aux objets implicites comme out, request, response, session, application, etc.*
3. *JspFragment getJspBody() : retourne une instance de JspFragment, qui au moyen de la méthode invoke() de cet objet, permet de récupérer ou lancer l'interprétation du contenu des balises si ces dernières disposent de ${...} ou d'autres balises imbriquées.*

Nous solliciterons très souvent la méthode getJspContext(), puisque la pluspart du temps, nous avons besoin de travailler avec le contexte de la page. Cette méthode délivre une instance de la classe JspContext. Voici quelques méthodes utiles de cette classe :

1. *JspWriter getOut() : délibre le flux de sortie de la page JSP. représente l'équivalent de l'objet implicite out.*
2. *Object getAttribute() : récupère un attribut défini soit dans la page JSP, soit défini dans la session, soit défini dans l'application Web, etc. Il est donc possible de travailler directement avec un bean.*
3. *void setAttribute() : Permet de créer un nouvel attribut pour la page, la session, l'application, etc.*
4. *void removeAttribut() : Permet de supprimer un attribut de la page, de la session, de l'application, etc.*
5. *Object findAttribute() : retourne l'attribut défini en paramètre. Dans le cas contraire, renvoie null.*



Toutefois, cette classe JspContext, bien que très utilse, n'est pas suffisamment compétente, puisqu'elle ne concerne que la page elle-même. Il serait intéressant de travailler certe avec les éléments de la page mais également avec tout ce qui l'entoure, comme les sessions, les configurations, les requêtes, etc.

Il existe effectivement une classe plus performante que nous avons déjà rencontré dans l'étude sur les expressions EL. Il s'agit de la classe PageContext. Cette classe hérite de la classe JspContext et permet donc de travailler au moins comme elle en récupèrant toutes les méthodes que nous venons de découvrir.

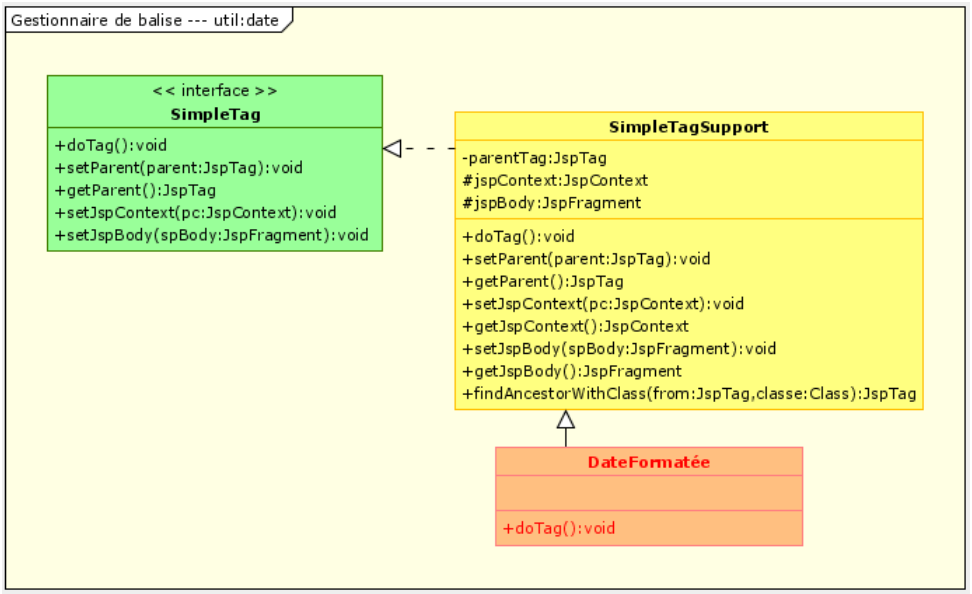
L'intérêt de cette classe PageContext, c'est que, en plus des méthodes que nous venons de voir, elle possède des constantes et des méthodes qui permettent de travailler directement avec les objets implicites out, request, response, session, application, exception, etc.

Du coup, il est souvent judicieux, lorsque nous faisons appel à la méthode getJspContext() de transtyper le retour vers un PageContext, puisque cette méthode délivre normalement un JspContext. Il faut savoir toutefois que l'objet construit dans la classe SimpleTagSupport est bien réellement de type PageContext puisque la classe JspContext est abstraite.

Au travers des exemples ci-desssous, vous allez découvrir l'ensemble des possibilités de ces méthodes et classes.

Gestionnaire de la balise <util:date>

A titre d'exemple, nous allons mettre en oeuvre le gestionnaire de balise correspondant à l'action personnalisée <util:date>. Nous appelons la classe représentant cette balise : DateFormatée. Cette action personnalisée ne possède pas de corps, ni d'attribut. Elle est donc facile à implémenter. Il suffit de redéfinir uniquement la méthode doTag() qui s'occupera de spécifier l'affichage de la date en format français.



DateFormatée.java

```
1 package util;
2
3 import java.io.IOException;
4 import javax.servlet.jsp.tagext.*;
```

```java
import javax.servlet.jsp.*;

import java.util.Date;
import java.text.DateFormat;

public class DateFormatée extends SimpleTagSupport {

   public void doTag() throws JspException, IOException {
      JspWriter pageJSP = getJspContext().getOut();
      DateFormat formatDate = DateFormat.getDateInstance(DateFormat.FULL);
      pageJSP.print(formatDate.format(new Date()));
   }
}

```

Ligne 3 à 5 :

Importation minimale pour développer un gestionnaire de balise.

Ligne 10 :

Gestionnaire **DateFormatée** qui hérite de la classe **SimpleTagSupport**.

Lignes 12 à 16 :

Exécution de tout ce que doit réaliser la balise **<util:date>**. La méthode **doTag()** accède au flot HTML par les deux appels **getJspContext().getOut()**.

Lignes 13 :

Récupération du contexte d'affichage de la page JSP qui utilise la balise **<util:date>**. La méthode **getJspContext()** délivre un objet de type **JspContext** qui représente la page JSP appelante. En général, c'est un objet représentant la page elle-même, mais à partir de la version 2.0, les tags sont conçus pour pouvoir être utilisés dans d'autres environnements qu'une page JSP. Cet objet possède, entre autre, une méthode **getOut()** qui renvoie un objet de type **JspWriter** qui représente le flot HTML. Grâce à ce flot de texte (**Writer**), nous pourrons envoyer notre propre texte dans la page Web créée.

Ligne 14 :

Réglage du formatage de la date.

Ligne 15 :

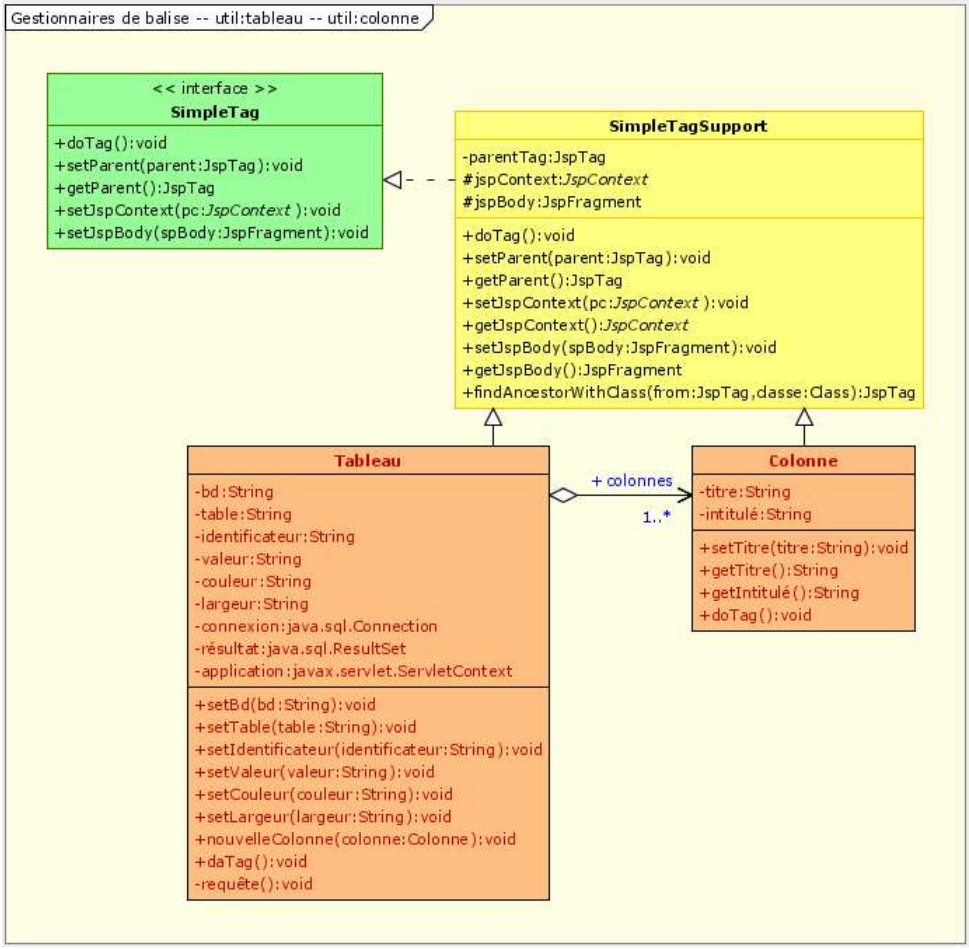
Instanciation de la date actuelle, et envoie de sa représentation formatée sous forme de chaîne de caractères dans le flot de sortie.

Par cet exemple, vous remarquez que la création d'un gestionnaire de balise simple est très facile à mettre en oeuvre.
.

Gestionnaires des balises <util:tableau> et <util:colonne>

```
<util:tableau bd="messagerie" table="message" identificateur="idPersonne" valeur="${idPersonne}" couleur="#FF6600" largeur="90%">
  <util:colonne titre="sujet">Sujet</util:colonne>
  <util:colonne titre="texte">Message correspondant</util:colonne>
</util:tableau>
```

*Vu que nous avons deux balises à traiter, nous allons également définir deux gestionnaires de balise, c'est-à-dire les deux classes représentant chacune des balises respectivement, la classe **Tableau**, la classe **Colonne**. Par ailleurs, ces deux classes sont en relation puisque leurs balises respectives sont imbriquées. Ainsi, la classe **Colonne** peut connaître la classe **Tableau** englobante au moyen de la méthode **getParent()**. Il suffit alors pour la classe **Tableau** de développer des méthodes spécifiques afin de permettre la récupération des informations de la classe **Colonne**.*



La première classe Tableau est un peu volumineuse. C'est un peu normal puisque la balise <:tableau> dispose de pas mal d'attributs que cette classe doit gérer. Par ailleurs, elle doit s'occuper également de la gestion des colonnes. Enfin, cette classe est en relation directe avec la base de données et propose des requêtes adaptées au besoin de l'utilisateur. Cet investissement vaut le coup, puisque ce gestionnaire pourra être déployé sur d'autres applications Web.

Tableau.java

```
 1 package util;
 2
 3 import java.io.IOException;
 4 import java.util.ArrayList;
 5 import javax.servlet.*;
 6 import javax.servlet.jsp.tagext.*;
 7 import javax.servlet.jsp.*;
 8 import java.sql.*;
 9
10 public class Tableau extends SimpleTagSupport {
11    private String bd;
12    private String table;
13    private String identificateur;
14    private String valeur;
15    private String couleur;
16    private String largeur;
17
18    private Connection connexion;
19    private ResultSet résultat;
20    private ServletContext application;
21
22    public void setBd(String bd) { this.bd = bd; }
23    public void setTable(String table) { this.table = table; }
24    public void setIdentificateur(String identificateur) { this.identificateur = identificateur; }
25    public void setValeur(String valeur) { this.valeur = valeur; }
26    public void setCouleur(String couleur) { this.couleur = couleur; }
27    public void setLargeur(String largeur) { this.largeur = largeur; }
28
29    private ArrayList<Colonne> colonnes = new ArrayList<Colonne>();
30    public void nouvelleColonne(Colonne colonne) { colonnes.add(colonne); }
31
32    public void doTag() throws JspException, IOException {
33       application = ((PageContext)getJspContext()).getServletContext();
34       if (bd==null) bd = application.getInitParameter("bd");
35       if (largeur==null) largeur = "100%";
36
37       JspWriter out = getJspContext().getOut();
38       this.getJspBody().invoke(null);
39
40       out.println("<table border=\"1\" cellpadding=\"3\" cellspacing=\"2\" width=\""+largeur+"\" align=\"center\">");
41       out.println("<tr bgcolor=\""+couleur+"\">");
42       for (int i=0; i<colonnes.size(); i++)
43          out.println("<th>"+colonnes.get(i).getIntitulé()+"</th>");
44       out.println("</tr>");
45       try {
46          requête();
47          while (résultat.next()) {
48             out.println("<tr>");
49             for (int i=0; i<colonnes.size(); i++)
50                out.println("<td>"+résultat.getString(colonnes.get(i).getTitre())+"</td>");
51             out.println("</tr>");
52          }
53          connexion.close();
54       }
55       catch (Exception erreur) { out.println("<tr><td>Service inactif pour l"instant</td></tr>"); }
56       out.println("</table>");
57    }
58
59    private void requête() throws SQLException, ClassNotFoundException {
60       String pilote = application.getInitParameter("pilote");
61       String localisation = application.getInitParameter("localisation");
62       String utilisateur = application.getInitParameter("utilisateur");
63       String motDePasse = application.getInitParameter("motDePasse");
64
65       Class.forName(pilote);
66       connexion = DriverManager.getConnection(localisation+bd, utilisateur, motDePasse);
67       Statement instruction = connexion.createStatement();
68       résultat = instruction.executeQuery("SELECT * FROM "+table+" WHERE "+identificateur+"=\""+valeur+"\"");
69    }
70 }
71
```

Lignes 11 à 16 et 22 à 27 :

La prise en compte des attributs de la balise **<:tableau>** se fait à ce niveau là. En effet, il suffit de mettre en place des propriétés portant le même nom que les attributs de la balise. C'est simple, les attributs de la classe sont les attributs de la balise correspondante. Par ailleurs, pour que ces attributs soient considérés comme des propriétés, il en plus nécessaire de mettre en place des méthodes accesseurs. Dans le cas des gestionnaires de balise, la plupart du temps, seules les méthodes de type **setXxx()** sont nécessaires.

Lignes 18 à 20 :

Bien entendu, il est souvent utile de rajouter quelques attributs supplémentaires afin depermettre la communication entre les différentes méthodes de la classe.

Lignes 29 et 30 :

Mise en place de la liste des colonnes. C'est la classe **Colonne**, au moyen de la méthode **nouvelleColonne()**, qui va compléter cette liste.

Lignes 32 à 57:

Exécution de tout ce que doit réaliser la balise **<util:Tableau>**.

Ligne 33 :

Création de l'objet application correspondant tout-à-fait à l'objet implicite application que nous avons déjà vu. C'est d'ailleurs pour cette raison que je lui est donné ce nom là. Grâce à cet objet, il sera donc possible de récupérer toutes les informations propres à l'application Web. Rappelez-vous que ces informations

sont décrites dans le descripteur de déploiement, ce qui permet de proposer des configurations différentes (changement de serveur de base de données, changement de pilote, etc.).

Ligne 34 :

Justement, vu que le nom de la base de données est un attribut optionnel, et dans le cas où il n'est pas précisé, il doit alors être recherché dans le descripteur de déploiement.

Ligne 35 :

La largeur est également optionnelle, il faut donc prévoir une dimension par défaut.

Ligne 37 :

Comme pratiquement, dans tous les cas, nous avons besoin du flux de sortie de la page JSP, afin de proposer l'affichage du tableau.

Ligne 38 :

Cette ligne est importante. Avant d'effectuer les traitements proprement dit, il est nécessaire de connaitre les balises **<:colonnes>** qui sont imbriquées dans la balise **<:tableau>**. Ainsi, nous serons capable de construire le tableau correctement. Il faut pour cela que la page JSP interprète la suite des instructions qui se trouve à l'intérieur de la balise **<:tableau>**, c'est-à-dire le corps de la balise. Comme nous l'avons appris, il suffit donc de faire appel à la méthode **invoke()** de la classe **JspFragment**. L'objet de cette classe est délivré par la méthode **getJspBody()**. Le paramètre de la méthode **invoke()** est **null** ce qui signifie que nous récoltons les informations de la page JSP elle-même. L'interprétation du corps de cette balise consiste à créer des beans **Colonne** qui entrerons en relation avec le bean parent **Tableau**. Ainsi, à l'issu de cette simple instruction, le bean **Tableau** connaitra parfaitement l'ensemble des colonnes qu'il aura à sa disposition.

Ligne 40 à 44 :

Commencement de la construction du tableau avec la récupération de l'intitulé de chacune des colonnes présentes.

Lignes 45 à 56 :

Traitement de la requête définie plus loin. Le tableau est complété par les informations issues de cette requête.

Lignes 59 à 69 :

Méthode établissant la communication avec la base de données.

Ligne 60 à 63 :

Récupération de toutes les informations définies dans le descripteur de déploiement.

Lignes 65 à 68 :

Mise en place de la requête demandée par l'utilisateur et récupération des informations nécessaires afin d'en permettre l'exploitation dans la méthode **doTag ()**.

Vous avez ci-dessous une partie du descripteur de déploiement décrivant les paramètres de configuration de l'ensemble de l'application Web utilisés par la classe Tableau.

extrait de web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4" 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 http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
...
...
  <context-param>
    <param-name>couleurFond</param-name>
    <param-value>FFFF66</param-value>
  </context-param>
  <context-param>
    <param-name>pilote</param-name>
    <param-value>com.mysql.jdbc.Driver</param-value>
  </context-param>
  <context-param>
    <param-name>localisation</param-name>
    <param-value>jdbc:mysql://localhost/</param-value>
  </context-param>
  <context-param>
    <param-name>bd</param-name>
    <param-value>messagerie</param-value>
  </context-param>
  <context-param>
    <param-name>utilisateur</param-name>
    <param-value>root</param-value>
  </context-param>
  <context-param>
    <param-name>motDePasse</param-name>
    <param-value>manu</param-value>
  </context-param>

</web-app>
```

La classe Colonne suivante représente la balise <:colonne> qui est imbriquée dans la balise <:tableau>. Cette balise <:colonne> dispose d'un seul attribut obligatoire qui indique quel est la colonne de la table issue de la base de données à afficher. Ensuite, le contenu de cette balise précise quel est l'intitulé de la colonne que nous afficherons dans la partie en-tête du tableau (dans la balise <h>).

Colonne.java

```
 1 package util;
 2
 3 import java.io.*;
 4 import javax.servlet.jsp.tagext.*;
 5 import javax.servlet.jsp.*;
 6
 7 public class Colonne extends SimpleTagSupport {
 8    private String titre;
 9    private String intitulé;
10
11    public void setTitre(String titre) {
12       this.titre = titre;
```

```
13    }
14    public String getTitre() {
15       return titre;
16    }
17    public String getIntitulé() {
18       return intitulé;
19    }
20
21    public void doTag() throws JspException, IOException {
22       StringWriter corps = new StringWriter();
23       this.getJspBody().invoke(corps);
24       intitulé = corps.toString();
25       ((Tableau) this.getParent()).nouvelleColonne(this);
26    }
27 }
28
```

Ligne 8 et 11 à 16 :

Définition de la propriété **titre** en relation avec l'attribut correspondant de la balise **<:colonne>**. Cette fois-ci, les deux accesseurs **getTitre()** et **setTitre()** sont sollicitées. La méthode **getTitre()** sera utile pour la classe **Tableau**.

Lignes 9 et 17 à 19:

Attribut et méthode de lecture également utile pour la classe **Tableau** afin de permettre la récupération de l'intitulé de la colonne.

Lignes 21 à 26 :

Pour chaque balise **<:colonne>**, il est nécessaire de récupérer le contenu du corps de la balise. Dès que nous faisons référence au corps d'une balise, nous devons faire appel à la méthode **invoke()** de l'objet représentant un **JspFragment** délivré par la méthode **getJspBody()**. Cette méthode **invoke()** attend un argument de type **Writer**. Ici, il s'agit de récupérer l'information sans l'afficher ou la traiter dans la page JSP elle-même. L'astuce consiste à créer un **StringWriter** de type flux de chaîne de caractères, ainsi l'information est tout simplement stockée en mémoire. Il suffit ensuite de récupérer cette chaîne de caractères afin de définir correctement l'attribut **intitulé**. Vu que les attributs de l'objet représentant la classe **Colonne** sont parfaitement définis, nous pouvons maintenant envoyer la totalité de ces informations à l'objet représentant la classe **Personne**, et ceci au moyen de la méthode **getParent()**.

Gestionnaire de la balise <util:date> : 2ème édition

Nous allons revenir sur le gestionnaire de balise <:date>. Nous changeons son comportement. En effet, cette fois-ci, c'est le webmaster qui indique comment il vaut formatter la date. Le but de ce gestionnaire est tout simplement de créer de nouveaux attributs pour la page JSP (qui seront valides d'ailleurs sur la totalité de la page). En elle-même la balise <:date> n'affiche plus la date. Il faut faire appel à (ou aux) attributs concernés. Le gestionnaire de balise <:date> fabrique les attributs suivant :

1. ***jourSemaine** : lundi, mardi, mercredi, etc.*
2. ***jourMois** : 1..31;*
3. ***mois** : janvier, février, mars, etc.*
4. ***année** : 2005.*
5. ***date** : vendredi 16 décembre 2005.*

DateFormatée.java

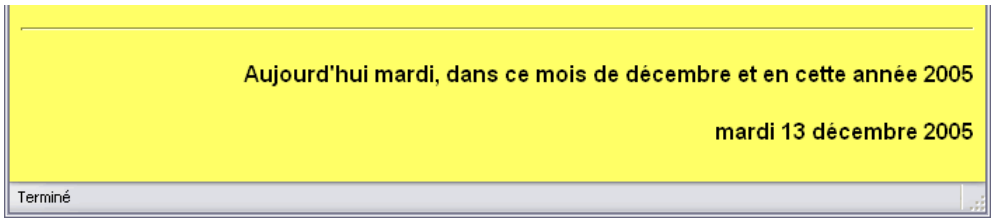
```
1  package util;
2
3  import java.io.*;
4  import javax.servlet.jsp.tagext.*;
5  import javax.servlet.jsp.*;
6
7  import java.util.Date;
8  import java.text.*;
9
10 public class DateFormatée extends SimpleTagSupport {
11
12    public void doTag() {
13       DateFormat formatDate = DateFormat.getDateInstance(DateFormat.FULL);
14       Date date = new Date();
15       String jourSemaine = new SimpleDateFormat("EEEE").format(date);
16       String jourMois = new SimpleDateFormat("d").format(date);
17       String mois = new SimpleDateFormat("MMMM").format(date);
18       String année = new SimpleDateFormat("yyyy").format(date);
19       this.getJspContext().setAttribute("date", formatDate.format(date));
20       this.getJspContext().setAttribute("jourSemaine", jourSemaine);
21       this.getJspContext().setAttribute("jourMois", jourMois);
22       this.getJspContext().setAttribute("mois", mois);
23       this.getJspContext().setAttribute("année", année);
24    }
25 }
26
```

Pour créer les attributs de page, il suffit de faire appel à la méthode setAttribute() d'un objet de type JspContext (ou PageContext). Le premier argument de cette méthode est le nom de l'attribut qui pourra être utilisé par la page JSP. Le deuxième argument est l'objet correspondant à l'attribut.

Voici une utilisation possible :

```
1 <%@taglib uri="/util" prefix="u" %> <u:date />
2
3      <br><hr>
4      <h4 align="right">
5        Aujourd"hui ${jourSemaine}, dans ce mois de ${mois} et en cette année ${année}
6      </h4>
7      <h4 align="right">${date}</h4>
8    </font>
9   </body>
10 </html>
11
```

Dont vous avez le résultat ci-dessous :



Contrôle du type d'authentification <:authentification> de notre application Web

Ce gestionnaire de balise va nous permettre d'authentifier l'utilisateur. Ce gestionnaire ne sera utile que pour cette application Web. Du coup, je le place dans un paquetage appelé 'personnel'. Au moyen de ce gestionnaire, nous allons découvrir qu'il est facile de faire référence aux objets implicites. Ici, nous avons besoin des objets suivants : pageContext, out, session, request.

Seule la méthode doTag() est redéfinie. Du coup, nous nous trouvons dans un schéma classique, et je ne représente donc pas le diagramme UML correspondant.

Authentification.java

```
 1 package personnel;
 2
 3 import java.io.IOException;
 4 import javax.servlet.*;
 5 import javax.servlet.http.*;
 6 import javax.servlet.jsp.tagext.*;
 7 import javax.servlet.jsp.*;
 8
 9 public class Authentification extends SimpleTagSupport {
10
11    public void doTag() throws JspException, IOException {
12       PageContext pageContext = (PageContext) this.getJspContext();
13       JspWriter out = pageContext.getOut();
14       HttpSession session = pageContext.getSession();
15       ServletRequest request = pageContext.getRequest();
16 //       ServletRequest request = (ServletRequest)getJspContext().getAttribute(PageContext.REQUEST);
17
18       session.removeAttribute("utilisateur");
19       String authentification = request.getParameter("authentification");
20       boolean nouveau = authentification.equals("nouveau");
21       out.print(nouveau?"Demande d'inscription":"Vos références");
22       this.getJspContext().setAttribute("prochainePage", nouveau ? "valider" : "controle");
23    }
24 }
25
```

Certainement, la méthode la plus utilisée de SimpleTagSupport est getJspContext(). Effectivement, nous avons toujours besoin du contexte de la page. Toutefois, il est souvent très judicieux de faire un transtypage, dès le départ, vers un PageContext afin de disposer de toutes les méthodes et constantes supplémentaires. Nous avons vu, en effet, que cette classe hérite de la classe JspContext et dispose des méthodes spécifiques qu'il suffit de choisir pour nous connecter vers le bon objet implicite - getOut(), getSession(), getRequest().

Une autre solution, mais souvent plus longue, consiste à utiliser la méthode getJspContext() sans transtypage. Au moyen de cette méthode, nous faisons ensuite appel à la méthode getAttribute() et là, nous précisons en argument de la méthode, la constante désirée donnée par la classe PageContext. (ligne 16).

Vérification de l'identité de l'utilisateur <:verifier>

Le dernier gestionnaire de balise que nous allons mettre en oeuvre est juste là à titre pédagogique. Son intérêt peut être discutable. Toutefois, la balise correspondante <:verifier />, est capable de récupérer les valeurs d'un bean, d'être en relation avec la session, et enfin de rediriger la requête vers la page d'accueil <bienvenue.jsp>.

Ainsi, toute la partie contrôle d'identité, suppression de l'attribut utilisateur dans la session, et redirection au moyen de l'action <jsp:forward /> va être résolu au moyen d'une seule balise <:verifier /> dont voici l'écriture ci-dessous :

controleidentite.jsp

```
1 <%@ page errorPage = "/WEB-INF/erreur.jsp" import="bd.*" %>
2 <%@taglib uri="/personnel" prefix="personnel" %>
3
4 <jsp:useBean id="utilisateur" class="bd.Personne" scope="session">
5     <jsp:setProperty name="utilisateur" property="*" />
6 </jsp:useBean>
7
8 <personnel:verifier />
```

Voici ci-dessous le gestionnaire de balise correspondant. Encore une fois, seule la méthode doTag() est redéfinie. Je ne présenterais donc pas le diagramme UML équivalent.

Verifier.java

```
 1 package personnel;
 2
 3 import bd.Personne;
 4 import java.io.IOException;
 5 import java.sql.SQLException;
 6 import javax.servlet.ServletException;
 7 import javax.servlet.http.HttpSession;
 8 import javax.servlet.jsp.tagext.*;
 9 import javax.servlet.jsp.*;
10
11 public class Verifier extends SimpleTagSupport {
12
13    public void doTag() throws JspException, IOException {
14       PageContext pageContext = (PageContext) this.getJspContext();
15       JspWriter out = pageContext.getOut();
16       HttpSession session = pageContext.getSession();
17       Personne utilisateur = (Personne) pageContext.getAttribute("utilisateur", PageContext.SESSION_SCOPE);
18       try {
```

```
19          if (!utilisateur.authentifier()) {
20             session.removeAttribute("utilisateur");
21          }
22          pageContext.forward("Demarrage/bienvenue");
23       }
24       catch (SQLException ex) { ex.printStackTrace(); }
25       catch (ServletException ex) { ex.printStackTrace(); }
26    }
27 }
28
```

Nous avons la même approche que pour le gestionnaire précédent. Nous récupérons tout de suite un objet pageContext de la classe PageContext. Ainsi, nous pouvons travailler plus facilement, afin d'utiliser les méthodes adaptées à la situation.

Pour récupérer le bean 'utilisateur' créé dans la page JSP, encore une fois, il faut faire appel à la méthode getAttribute(), mais cette fois-ci, celle qui possède deux arguments. Le deuxième argument sert à préciser dans quel scope se situe le bean, ici la session.

Nous pouvons rediriger très simplement la requête prévue pour cette page puisque la classe PageContext dispose de la méthode forward() que nous connaissons bien.

Je crois que nous avons fait un bon tout d'horizon, et je pense qu'au moyen de ces exemples, nous pouvons traiter un bon nombre de gestionnaire de balise. Essayez toutefois toutes les méthodes que vous avez à votre disposition. Vous découvrirez ainsi plusieurs autres applications possibles.

Descripteur de bibliothèque de balises

Après avoir créé une ou plusieurs classes implémentant une action, nous devons indiquer au conteneur quels gestionnaires de balises sont disponibles pour traiter les pages de l'application. C'est le rôle du descripteur de bibliothèque de balises ou TLD (Tag Library Descriptor).

Le TLD est un document XML contenant des informations concernant les gestionnaires de balises disponibles dans une bibliothèque. Un TLD compatible JSP 2.0 fournit des informations concernant une biblothèque de balises au moyen d'un élément <taglib>.

L'élément <taglib> peut avoir un certain nombre de sous-éléments. Les sous-éléments obligatoires sont les suivant :

Elément	Signification
<tlib-version>	Le numéro de version de la bibliothèque
<short-name>	Un nom employé par défaut. Il peut être utilisé comme préfixe dans les directives taglib. Notez, toutefois, que les développeurs de pages JSP peuvent employer le préfixe de leur choix. De cette façon, il est facile d'éviter tout conflit de nom avec d'autres bibliothèques. Cet élément <short-name> peut éventuellement être vide
<tag>	Informations diverses concernant le gestionnaire de balises (Composé d'un certain nombre de sous-éléments dont <name> et <tag-class>).
<name>	Le nom du gestionnaire de balise. C'est là que nous définissons le nom de notre balise personnalisée. Le choix de ce nom est important.
<tag-class>	Le nom qualifié de la classe implémentant le gestionnaire. C'est la classe qui implémente le comportement de notre balise personnalisée.

Il existe un autre sous-élément de <taglib>, non obligatoire, qui me paraît pourtant bien utile :

Elément	Signification
<uri>	Il s'agit du nom de l'URI à préciser dans la page JSP, afin que cette dernière puisse importer cette bibliothèque de balises au travers de la directive <% @ taglib uri="/NomUri" ... %>.

Par ailleurs, les sous-éléments de <tag> optionnels suivants sont très souvent indispensables :

sous-élément	Signification
<body-content>	Indique si l'action peut avoir un contenu. Les valeur possible sont : 1. *tagdependent : Le corps du tag ne sera pas interprété. Si il contient du code il sera affiché comme du simple texte,* 2. *scriptless : Le corps du tag ne peut alors contenir que du texte, des expressions EL et d'autres tags JSP, mais aucun scripts Java (<% ... %>).* 3. *empty : La balise n'accepte aucun corps (une exception sera lancée, si il est utilisé avec un corps quelconque).*
<variable>	Définit les variables de script créées par ce gestionnaire et mises à la disposition de la page. Cet élément doit contenir un des deux sous-éléments suivants <name-given> ou <name-from-attribute>. Si <name-given> est présent, la valeur de cet élément définit le nom qui doit être utilisé par les autres éléments JSP de la page pour accéder à la variable de script. En revanche, <name-from-attribute> indique que la valeur de l'attribut indiqué par cet élément est le nom de la variable de script.
<attribut>	Définit les attributs de la balise. Cet élément possède trois sous-éléments : <name>, <required> et <rtexprvalue>. 1. *La valeur de l'élément <name> est le nom de l'attribut.* 2. *L'élément <required> est optionnel et peut prendre les valeurs true, false, yes ou no. La valeur par défaut est false.* 3. *L'élément <rtexprvalue> est optionnel et peut prendre les valeurs true, false, yes ou no. La valeur par défaut est false, ce qui signifie que l'attribut ne peut recevoir une valeur que si celle-ci est statique et connue au moment de la compilation. Si la valeur est true ou yes, l'attribut peut recevoir une valeur dynamique au moment de l'exécution.*

Pour bien comprendre et maîtriser le rôle de chaque élément, je vous propose de l'expliquer au travers des exemples précédents.

Descripteur de balise (TLD) associé à la balise <util:date>

Voici donc le descripteur qui va permettre l'utilisation de la balise <:date>. Il sera placé dans le répertoire <tlds> prévu à cet effet dans la zone privée <WEB-INF> de l'application Web. Par ailleurs, l'extension de ce fichier est <.tld>.*

/WEB-INF/tlds/util.tld

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <taglib version="2.0" xmlns="http://java.sun.com/xml/ns/j2ee"
3         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4         xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee web-jsptaglibrary_2_0.xsd">
5
6   <tlib-version>1.0</tlib-version>
7   <short-name>util</short-name>
```

```
 8   <uri>/util</uri>
 9
10   <tag>
11       <name>date</name>
12       <tag-class>util.DateFormatée</tag-class>
13       <body-content>empty</body-content>
14   </tag>
15 </taglib>
16
```

Ligne 1 à 4 :

Ecriture systématique qui correspond à la déclaration de la bibliothèque. Ce type d'écriture est obligatoire lorsque nous mettons en place un document XML. Vous pouvez donc faire du copier-coller si vous utilisez la spécification JSP 2.0. Ceci dit, beaucoup d'éditeurs de système de développement proposent des écritures automatiques.

Ligne 6 :

Balise obligatoire. A l'heure actuelle, toujours la version 1.0.

Lignes 7 :

Balise obligatoire qui spécifie l'espace de nom par défaut.

Lignes 8 :

Balise non obligatoire qui spécifie le nom de l'URI à placer dans l'attribut **uri** de la directive **taglib** : **<%@ taglib uri="/util" ... %>**.

Ligne 10 à 14:

Définition d'une balise.

Ligne 11 :

Nom de la balise. Ici, nous choisissons, bien évidemment, **date**.

Ligne 12 :

Gestionnaire de balise associé, c'est-à-dire, la classe qui implémente l'actionde cette balise.

Ligne 13 :

Nous indiquons ici que la balise ne doit pas avoir un corps. Cela interdit donc la syntaxe suivante **<:date>Du texte</:date>**. Ce genre d'écriture n'a aucun sens dans ce contexte. Bien que cette balise **<body-content>** ne soit pas obligatoire, il me paraît important de préciser comment doit agir notre action personnalisée afin d'éviter de mauvaises utilisations ultérieures.

Lorsque nous avons une balise simple à implémenter, vous remarquez que le TLD est également simple à mettre en oeuvre. En gros, nous avons juste une association entre le nom de la balise (dont le nom est à notre libre initiative) et la classe qui réalise le traitement correspondant.

Rajout de <util:tableau> et <util:colonne> dans le descripteur de balises précédent

Nous rajoutons au descripteur de balises <util.tld> la définition des balises <:tableau> et <:colonne>. Cette fois-ci, le descripteur comporte beaucoup plus d'informations. En effet, la balise <:tableau> notamment comporte beaucoup d'attributs. C'est ce qui prend beaucoup de place.

/WEB-INF/tlds/util.tld

```
 1 <?xml version="1.0" encoding="UTF-8"?>
 2 <taglib version="2.0" xmlns="http://java.sun.com/xml/ns/j2ee"
 3     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 4     xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee web-jsptaglibrary_2_0.xsd">
 5
 6   <tlib-version>1.0</tlib-version>
 7   <short-name>util</short-name>
 8   <uri>/util</uri>
 9
10   <tag>
11     <name>date</name>
12     <tag-class>util.DateFormatée</tag-class>
13     <body-content>empty</body-content>
14   </tag>
15   <tag>
16     <name>tableau</name>
17     <tag-class>util.Tableau</tag-class>
18     <body-content>scriptless</body-content>
19     <attribute>
20       <name>bd</name>
21       <required>false</required>
22       <rtexprvalue>true</rtexprvalue>
23     </attribute>
24     <attribute>
25       <name>table</name>
26       <required>true</required>
27       <rtexprvalue>true</rtexprvalue>
28     </attribute>
29     <attribute>
30       <name>identificateur</name>
31       <required>true</required>
32       <rtexprvalue>true</rtexprvalue>
33     </attribute>
34     <attribute>
35       <name>valeur</name>
36       <required>true</required>
37       <rtexprvalue>true</rtexprvalue>
38     </attribute>
39     <attribute>
40         <name>couleur</name>
41         <required>false</required>
42         <rtexprvalue>true</rtexprvalue>
43     </attribute>
44     <attribute>
45         <name>largeur</name>
46         <required>false</required>
```

```
47          <rtexprvalue>true</rtexprvalue>
48      </attribute>
49    </tag>
50    <tag>
51      <name>Colonne</name>
52      <tag-class>util.Colonne</tag-class>
53      <body-content>tagdependent</body-content>
54      <attribute>
55        <name>titre</name>
56        <required>true</required>
57        <rtexprvalue>true</rtexprvalue>
58      </attribute>
59    </tag>
60  </taglib>
61
```

Ligne 10 à 49 :

Définition de la balise **<:tableau>**.

Lignes 18 :

Cette fois-ci le corps de la balise n'est pas vide. Elle devra effectivement comporter d'autres balises enfants **<:colonne>** qui devront donc être interprétées. Pour cela, il faut le préciser avec le mot **scriptless**.

Lignes 19 à 23 :

Définition du premier attribut. Il s'agit de l'attribut **bd** qui représente la base de données. Cet attribut n'est pas obligatoire. Par contre, il peut être défini à l'aide d'une variable si nous le désirons.

Lignes 24 à 48 :

Définition des autres attributs qui sont obligatoires sauf pour **couleur** et **largeur**.

Lignes 50 à 59 :

Définition de la balise **<:colonne>**.

Ligne 53 :

Cette fois-ci le corps de la balise doit être également présent mais sans interprétation particulière **tagdependent**.

Le descripteur de balises s'est considérablement allongé, mais sans difficultés particulières. C'est très souvent la définition des attributs qui prend de la place. En faisant du copier-coller, les descripteurs sont très simples à construire et nous n'y passons généralement pas beaucoup de temps.

Descripteur de balises (TLD) associé à <personnel:authentification> et <personnel:verifier>

Nous avons construit deux autres actions personnalisées. Elles sont particulières dans le sens où elles ne sont utiles que pour cette application Web. Du coup, j'ai préféré construire un autre descripteur de balise appelé <personnel.tld> indépendant du premier. En effet, <util.tld> qui comporte des balises suffisamment génériques pourra être déployé vers d'autres applications Web, alors que <personnel.tld> restera sur place.

/WEB-INF/tlds/personnel.tld

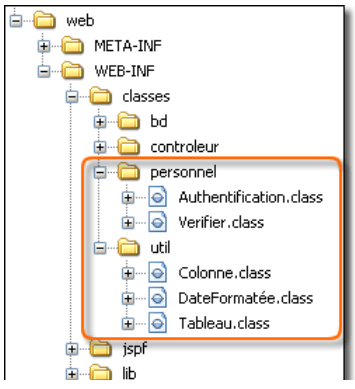
```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <taglib version="2.0" xmlns="http://java.sun.com/xml/ns/j2ee"
3     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4     xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee web-jsptaglibrary_2_0.xsd">
5
6     <tlib-version>1.0</tlib-version>
7     <short-name>personnel</short-name>
8     <uri>/personnel</uri>
9
10    <tag>
11      <name>authentification</name>
12      <tag-class>personnel.Authentification</tag-class>
13      <body-content>empty</body-content>
14    </tag>
15
16    <tag>
17      <name>verifier</name>
18      <tag-class>personnel.Verifier</tag-class>
19      <body-content>empty</body-content>
20    </tag>
21 </taglib>
22
```

Ce descriteur demeure simple et concis puisque chacune de ses balises <:authentifier> et <:verifier> ne comportent pas d'attributs et sont systématiquement vides.

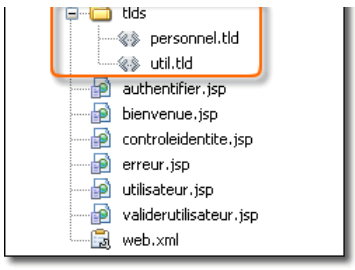
Structure de l'application Web pour intéger des actions personnalisées

Après avoir créé les classes du gestionnaire de balises et le TLD, il est maintenant important de les placer correctement dans l'application Web afin de respecter l'architecture prévue à cet effet. En respectant cette structure, le serveur d'application pourra automatiquement retrouver tous les composants nécessaires, et donc être à même de pouvoir répondre aux écritures (permettre les connexions adaptées) des actions personnalisées dans les pages JSP.

Dossiers	Type de fichier
webapp	*ressources Web publiques(pages accessibles directement)*



WEB-INF		*ressources Web privée(pages non accessibles directement)*
	classes	Gestionnaires de balises : util.DateFormatée.class util.Tableau.class util.Colonne.class personnel.Authentification.class personnel.Verifier.class
	tlds	Descripteurs de balises : util.tld personnel.tld
	lib	Bibliothèques de balises : <*.jar>
META-INF		Archives contenant des descripteurs de balises : <*.jar>



Les classes du gestionnaire de balises doivent être placées dans le sous-dossier </classes> du dossier <WEB-INF>, ou dans un fichier <.jar> se trouvant dans le dossier <WEB-INF/lib>. Dans notre exemple, un TLD est placé dans le dossier <WEB-INF/tlds>. Les descripteurs doivent être placés dans ce dossier particulier afin que le serveur d'application connaissent automatiquement les balises personnalisées à mettre en oeuvre. Si un TLD est placé dans un fichier <.jar>, celui-ci doit figurer dans le dossier <META-INF> de l'application.

Descripteur de déploiement

Il est possible (mais non obligatoire) de créer dans le descripteur de déploiement <web.xml>, une association entre une URI et l'emplacement d'un TLD. Il faut utiliser pour cela un élément <taglib>. Dans ce cas là, il n'est alors pas nécessaire de spécifier la balise <uri> dans le descripteur de balises. Cela ferait double emploi. Pour l'utilisation de notre balise <:date>, voilà ce qu'il faudrait écrire :

```
<taglib>
        <taglib-uri>/util</taglib-uri>
        <taglib-location>/WEB-INF/tlds/util.tld</taglib-location>
</taglib>
```

Ici, /util est associé au TLD util.tld. L'association peut ensuite être utilisée dans les pages JSP. Si vous décidez de ne pas utiliser le descripteur de déploiement pour réaliser cette association, il est alors nécessaire de le préciser directement dans le descripteur de balise, comme nous l'avons déjà vu, au moyen de la balise <uri>.

Importer une bibliothèque de balises et utilisation de ces balises dans la page JSP

Pour utiliser une action personnalisée, nous devons importer la bibliothèque de balises dans une page JSP, à l'aide de la directive taglib. Cette directive à la forme suivante :

```
<%@ taglib uri = "URI de la bibliothèque" prefix = "espace de nom" %>
```

Cette directive doit être placée avant toute utilisation d'une balise de la bibliothèque concernée. L'attribut uri peut être le chemin relatif ou absolu du TLD. Il peut également s'agir de l'URI associé à ce descripteur dans l'élément <taglib> du fichier <web.xml>. Enfin, et c'est souvent une bonne solution, l'attribut uri peut être le chemin précisé directement dans le descripteur de balises au niveau de la balise qui porte le même nom, <uri> :

```
<%@ taglib uri="/util" prefix="u" %>
```

Combinée à l'élément <uri> du descripteur de balises </WEB-INF/tlds/util.tld> de l'exemple précédent, cette directive importe la bibliothèque définie par le TLD <util.tld>. Dans la page contenant cette directive, les actions personnalisées peuvent être référencées au moyen du préfixe indiqué, comme ici u. Le TLD présenté précédemment dispose d'un gestionnaire de balise nommé date. Avec la directive ci-dessus, l'action peut être référencée dans une JSP par :

```
<u:date />
```

Voici d'ailleurs l'exemple complet du pied de page :

pieds.jspf

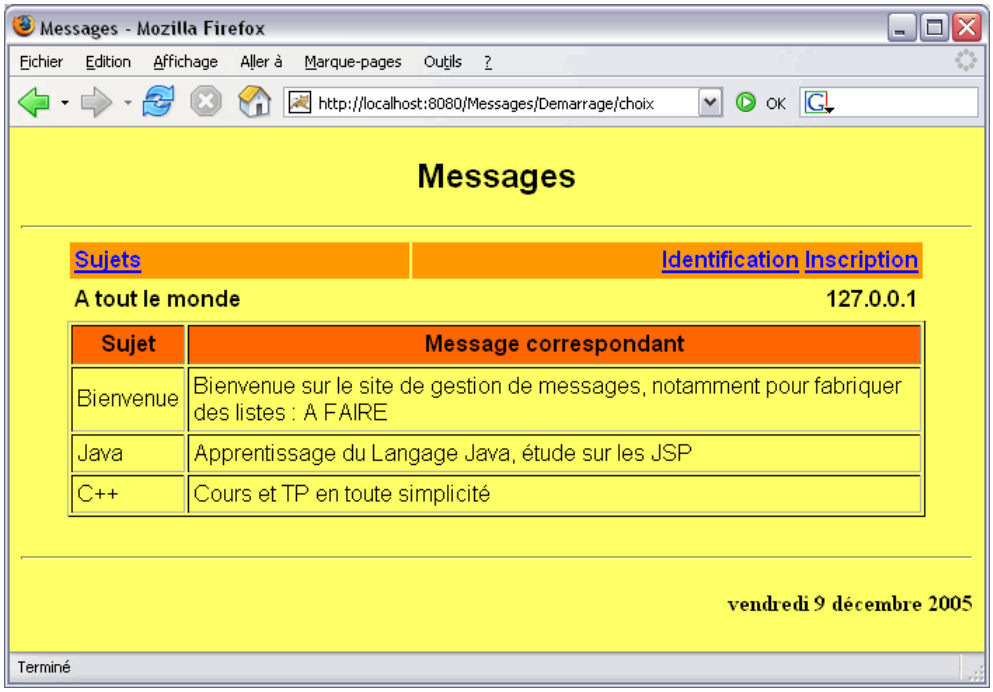
```
1 <%@taglib uri="/util" prefix="u" %>
2
3         <br><hr>
4         <h4 align="right"><u:date /></h4>
5      </font>
6    </body>
7 </html>
8
```

Voici également la page d'accueil qui utilise les actions personnalisées <:tableau>et <:colonne> :

bienvenue.jsp

```
 1 <%@ include file = "/WEB-INF/jspf/navigation.jspf" %>
 2 <%@taglib uri="/util" prefix="u" %>
 3
 4 <u:tableau table="message" identificateur="idPersonne" valeur="${idPersonne}" couleur="#FF6600" largeur="90%">
 5    <u:Colonne titre="sujet">Sujet</u:Colonne>
 6    <u:Colonne titre="texte">Message correspondant</u:Colonne>
 7 </u:tableau>
 8
 9 </font>
10
11 <%@ include file = "/WEB-INF/jspf/pieds.jspf" %>
```

Qui, en fin de compte, produisent le résultat suivant :



Encore une fois, nous avons passé quelques temps pour expliquer tout le déroulement à la mise en place des actions personnalisées. Vous remarquez que cela demeure, malgré tout, relativement simple. L'investissement vaut le coup, surtout si vous développez des balises suffisamment génériques pour être utilisées par d'autres applications Web. Ceci dit, rien n'empêche de fabriquer de nouvelles balises très spécialisées, même si elles ne seront utilisées que par une seule application Web. Le contenu des pages JSP gagne alors largement en clarté et en concision.

JSP Standard Tag Library - JSTL

Une grande partie de ce chapitre a été consacré à la création de balises personnalisées. Il n'est toutefois pas toujours indispensable de créer nos propres balises. Nous pouvons également employer de nombreuses bibliothèques disponibles. Dans cette section, nous porterons notre attention sur la JavaServer Pages Standard Tag Library - JSTL.

JSTL a été créée lorsque les responsables de J2EE se sont aperçus que de nombreux développeurs dépensaient beaucoup d'énergie pour créer des nouvelles balises répondant aux mêmes besoins. Etant développées séparément, les actions ainsi créées avaient des noms, des syntaxes et des comportements différents, mais elles accomplissaient pratiquement la même chose. Le but de JSTL est de standardiser un certain nombre d'actions parmi les plus fréquemment employées. De cette façon, vous pouvez passer d'une implémentation à une autre en changeant simplement le fichier <.jar> contenant la bibliothèque, mais sans avoir à modifier les pages JSP. Dans cette section, nous allons découvrir certaines des actions de la bibliothèque JSTL.*

Obtention d'une implémentation

Pour essayer JSTL, il suffit d'en obtenir une implémentation. Nous pouvons utiliser par exemple celle du projet jakarta, disponible à l'adresse :

http://jakarta.apache.org/taglibs/index.html

Son utilisation est très simple :

1. *Désarchivez le fichier contenant la distribution binaire que vous avez téléchargée depuis le site indiqué ci-dessus. Copier les fichiers <*.jar> dans le dossier </WEB-INF/lib> et les TLD dans le dossier </WEB-INF/tlds>.*
2. *Créez l'assosiation nécessaire dans le fichier <web.xml>.*
3. *Ajouter la directive taglib correspondante à vos pages JSP.*

Encore une fois, plutôt que de placer ces archives dans chacune des applications Web où nous devons les utiliser, il serait préférable de les placer directement dans le serveur Web. Toutefois, dans ce cas là, attention au déploiement, puisque ces archives ne font pas partie de votre application Web. Pour le serveur Tomcat, il faut placer les fichiers <standard.jar> et <jstl.jar> dans le répertoire <$TOMCAT_HOME/common/lib> ou <$TOMCAT_HOME/shared/lib>.

Contenu de la bibliothèque JSTL

Les actions JSTL sont réparties en quatre catégories, indiquées ci-après avec les TLD correspondants :

1. *actions fondamentales <c.tld> ;*
2. *traitement XML <x.tld> ;*
3. *mise en forme et internationalisation <fmt.tld> ;*
4. *accès aux bases de données relationnelles <sql.tld>.*

Fichiers descripteurs et directive taglib

La spécification 2.0 des pages JSP autorise le développeur à placer les fichiers descripteurs dans l'archive elle-même. Ainsi, les archives de la bibliothèque JSTL possèdent effectivement ces quatre descripteurs. Il ne sera donc pas nécessaire de s'en préoccuper. Toutefois, comme ils sont déjà construit, nous devons respecter l'URI désignée dans ces descripteur lors de l'utilisation des directives taglib. Ainsi, voici les écritures à préciser suivant la catégorie que nous devons utiliser :

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

Les actions fondamentales

Ces actions permettent de manipuler des variables, de traiter des conditions d'erreur, d'effectuer des tests et d'exécuter des boucles ou des itérations.

Les actions générales

Ces actions sont conçues pour manipuler les variables et traiter les conditions d'erreur.

Balise	Signification
<c:out value="" default="">	Envoie une valeur au stream de la réponse. Il est possible de spécifier une valeur par défaut pour le cas où la valeur est évaluée au travers d'une expression EL. Cette valeur sera utilisée si l'expression est évaluée à null. `6<c:out value='${param["x"]}' default="x absent" /> ` `7<c:out value='${c}' />` `8 <b>Pas d'attribut c dans les scopes actuels</b>` `9 </c:out>` Cette action remplace donc la balise <%=...%>, comme une simple instruction EL. Son avantage par rapport à une simple expression EL est l'utilisation d'une valeur par défaut et l'encodage des caractères XML réservés.
<c:set var="" value="" scope="">	Donne la valeur indiquée par l'attribut value à la variable dont le nom est spécifiée par l'attribut var de type String. L'attribut scope optionnel permet d'indiquer le scope désiré : page, request, session ou application. Par défaut, le scope est la page. `6<c:out value='${c}' default="Pas d'attribut c" /> ` `7<c:set value="coucou" var="c" scope="session" />` La première fois que ce code est exécuté, en supposant qu'il n'y ait aucun attribut de nom c dans aucun scope, le message 'Pas d'attribut' est produit puis la chaîne 'coucou' est rangée dans le scope session sous le nom c. La deuxième fois que le code est exécuté, la valeur de c est 'coucou' et s'affiche dans la page JSP. Dans l'exemple suivant, nous construisons un compteur au niveau de l'application Web : `6<c:set var="compteur" scope="application">` `7 <c:out value="${compteur+1}" default="1" />` `8</c:set>` `9 </c:out value="${compteur}" />` L'action <c:set> modifie la valeur de l'attribut compteur de l'application. Elle donne comme valeur à cet attribut la chaîne qu'elle contient. Celle-ci est le résultat de l'action <c:out> qui donne l'ancienne valeur du compteur augmentée de 1 si cette valeur existe, 1 si elle n'existe pas.
<c:set target="" property="" value="" >	Modifie la propriété d'un JavaBean ou d'une Map indentifié par target. `6<c:set target='${client}' property="nom" value='${request["nom"]} - ${request["prenom"]}' />` Si un scope possède l'attribut client et si cet attribut possède la méthode setNom(String s) alors cette méthode est invoquée avec pour paramètre effectif la concaténation du paramètre nom, d'un tiret et du paramètre prenom. *Attention, les mécanismes de résolution utilisés par <jsp:setProperty> et <c:set> ne sont pas indentiques. Il faut mettre la valeur de l'attribut target dans une expression EL, sinon l'objet désigné par l'attribut est une chaîne de caractères.*
<c:remove var="" scope="">	Supprime un objet identifié par le nom var dans la portée scope. L'attribut scope est optionnel. En son absence, toutes les portées sont examinées dans l'ordre page, request, session, application jusqu'à ce que l'objet soit trouvé. Si une portée est indiquée, l'objet est recherché seulement dans celle-ci. S'il n'est pas trouvé, une exception est lancée. `6<c:remove var="compteur" scope="application" />` Ces deux attributs ne peuvent pas avoir de valeur dynamique.
<c:catch var="">	Entoure un bloc de code susceptible de lancer une exception. Si l'exception est lancée, l'exécution du bloc se termine, mais l'exception n'est pas propagée. L'exception peut être référencée par la variable dont le nom est indiqué par l'attribut var. `<%--Si une exception survient ici, elle interrompt l'exécution de la page --%>` `<c:catch var="e">` `<!--Si une exception survient ici, elle interrompt l'exécution de l'élément mais pas celle de la page --!>` `</c:catch>` `<%--Si une exception survient ici, elle interrompt l'exécution de la page --%>` Seules les instructions non essentielles pour la page devraient se trouver dans une action <c:catch>. Si une telle instruction échoue, la page peut continuer à s'exécuter.

Les actions conditionnelles

Les actions conditionnelles permettent de tester les expressions et d'évaluer des balises en fonction du résultat.

Balise	Signification
<c:if test="" var="">	Cette action est employée comme un bloc if java. L'attribut var est optionnel. Le résultat du test est affecté à la variable identifiée par cet attribut. Si le résultat du test est true, la balise est évaluée. Dans le cas contraire, elle est ignorée. `6<c:if test="${jeu.gagne}">` `7 <!-- ... -->` `8</c:if>` Le contenu de l'action ne sera produit que si la méthode isGagne() renvoie la valeur true.

<c:choose> <c:when test=""> <c:otherwise>	Cette action est semblable au bloc if...elseif...else. La balise <c:choose> ouvre et termine le bloc. Le test indiqué par chaque élément <c:when test=""> est évalué. Le premier test dont la valeur est true entraine l'évaluation de la balise correspondante. Si aucun test ne renvoie true, la balise <c:otherwise> est évaluée. `<c:choose>` `  <c:when test="${jeu.gagne}">` `  <!-- ... -->` `  </c:when>` `  <c:when test="${jeu.perdu}">` `  <!-- ... -->` `  </c:when>` `  <c:otherwise>` `  <!-- ... -->` `  </c:otherwise>` `</c:choose>`

Les actions d'itération

Les actions d'itération permettent d'effectuer des boucles sur des ensembles de valeurs.

Balise	Signification
<c:forEach var="" items="">	Effectue une itération sur les éléments d'une collection identifiée par items. Chaque élément est référencé par var. Si items désigne une Map, la valeur d'un élément est référencée par var.value. L'attribut optionnel items permet de parcourir une collection non numérique. Dans le cas d'un tableau, la boucle se fait sur le contenu de ce tableau, de la première valeur à la dernière valeur par défaut. La variable définie par l'attribut var désigne successivement chacune des valeurs du tableau. Dans le cas d'une chaîne, celle-ci est considérée comme une énumération de valeurs séparées par des virgules. `<% string[] tableau = {"un", "deux", "trois"};` `  session.setAttribute("t", tableau};` `%>` `<ol>` `  <c:forEach var="v" items="${t}">` `  <li>${v}</li>` `  </c:forEach>` `</ol>` *Attention la valeur de l'attribut items est ${t} et non pas t. En effet, dans ce dernier cas, t serait considéré comme une chaîne de caractères constante constituée d'un seul caractère "t".*
<c:forEach var="" begin="" end="" step="">	Cette action permet d'effectuer une boucle for. L'attribut step est optionnel. `6<c:forEach begin="1" end="10" step="4" >` `7  Bonjour ` `8</c:forEach>` Ce code affiche trois fois le texte 'Bonjour ' : l'itération s'effectue avec la valeur 1, puis la valeur 5, puis la valeur 9. La valeur suivante, 13, est strictement supérieure à la dernière valeur admise pour l'itération qui est 10. Si l'attribut step est omis, sa valeur par défaut est 1. L'attribut optionnel var permet de céer une variable locale à l'action contenant la valeur de l'itération. `<table border="1">` `  <tr>` `  <c:forEach var="j" begin="1" end="10">` `  <th>${j}</th>` `  </c:forEach>` `  </tr>` `  <c:forEach var="i" begin="1" end="10">` `  <tr>` `  <th>${i}</th>` `  <c:forEach var="j" begin="1" end="10">` `  <td>${i*j}</td>` `  </c:forEach>` `  </tr>` `  </c:forEach>` `</table>` Table de multiplication sous forme de tableau.
<c:forTokens var="" items="" delims="">	Effectue une itération sur les mots d'une chaîne de caractères. L'action <c:forTockens> est semblable à l'action <c:forEach> utilisée avec une chaîne de caractères, mais nous pouvons choisir plusieurs délimiteurs au lien de la simple virgule grâce à l'attribut delims. `<ul>` `  <c:forTokens var="val" items='${header["accept-language"]}' delims=", ">` `  <li>${val}</li>` `  </c:forTockens>` `</ul>` Cette action produit la liste des langages acceptés par le navigateur, telle qu'envoyée dans l'en-tête de la requête.

Les actions de mise en forme

Les actions de mise en forme font partie de la bibliothèque I18N. Cette expression signifie Internationalisation (dix-huit lettres comprises entre i et le n). Nous trouvons dans cette bibliothèque des actions permettant de mettre en forme les nombres, les dates et les heures dans différentes langues.

Balise	Signification
<fmt:formatDate value="date" [type="{time|date|both}"] [dateStyle="{default|short|medium|long|full}"] [timeStyle="{default|short|medium|long|full}"] [pattern="schéma_personnalisé"] [timeZone="zone_horaire"] [var="nom_variable"] [scope="{page|request|session|application}"] / >	Seul l'attribut value est obligatoire. Les autres attributs définissent le format des données. L'attribut pattern permet de définir un schéma personnalisé pour la mise en forme des dates.

<fmt:formatNumber value="valeur_numérique" [type="{number|currency|percent}"] [pattern="schéma_personnalisé"] [currencyCode="code_monétaire"] [currencySymbol="symbole_monétaire"] [groupingUsed="{true|false}"] [maxIntegerDigits="max_chiffres_entiers"] [minIntegerDigits="min_chiffres_entiers"] [maxFractionDigits="max_chiffres_fraction"] [minFractionDigits="min_chiffres_fraction"] [var="nom_variable"] [scope="{page|request|session|application}"] / >	Cette action met en forme la valeur indiquée par l'attribut value. Différents styles sont disponibles, y compris les valeurs monétaires. Il est possible de définir ses propres formats. Cette action peut être employée sans l'attribut value. Dans ce cas, la mise en forme s'applique au contenu de la balise.
<fmt:setLocale value="code_pays" />	Cette action permet de sélectionner le pays pour lequel les données doivent être mise en forme. La valeur par défaut est us_US. Le code pour la france est fr_FR.

Les actions SQL

Les actions SQL permettent aux auteurs de pages d'effectuer des requêtes sur des bases de données, de consulter les résultats et de réaliser des insertions, des mises à jour et des suppressions.

Balise	Signification
<sql:query var="" sql="" dataSource="" scope="" > ....SQL Command </sql:query>	Cette balise sert à interroger la base de données définie par l'attribut dataSource. La requête est contenue dans le corps de la balise. Le résultat est accessible grâce à la variable var.rows. Il est possible d'utiliser l'action <c:forEach> pour parcourir les lignes du résultat. L'attribut dataSource peut identifier une base de données de deux façons différentes. Il peut s'agir d'une URL JDBC ou du nom JNDI d'une source de données. `<sql:query var="messages"` `dataSource="jdbc:mysql://127.0.0.1/messagerie,com.mysql.jdbc.Driver,root,manu"` `var="SELECT * FROM message" />` `<table>` `<c:forEach var="message" items="${messages.rows}">` `<tr>` `<td><c:out value="${message.sujet}"/></td>` `<td><c:out value="${message.texte}"/></td>` `</tr>` `</c:forEach>` `</table>`
<sql:param value="">	Afin d'optimiser les accès, les serveurs de base de données proposent la notion de requête préparée. Une telle requête est une requête incomplète dans laquelle certaines valeurs de champ sont remplacées par des variables. Cela permet au serveur de ne faire qu'un seul calcul de préparation de requête. Afin d'exécuter effectivement la requête, il faut fournir des valeurs aux différents paramètres. C'est le rôle de l'action <sql:param>. `<c:set id="personne" value="1"/>` `...` `<sql:query var="message" dataSource="jdbc:mysql://127.0.0.1/messagerie,com.mysql.jdbc.Driver,root,manu">` `SELECT * FROM message WHERE idPersonne = ?` `<sql:param value="${personne}"/>` `</sql:query>`
<sql:update var="" sql="" dataSource="" scope="" > ....SQL Command </sql:update>	Sans surprise, l'action <sql:update> effectue une modification de la base. La requête est fournie par l'attribut sql ou par le contenu de l'action. L'action admet les attributs dataSource, var et scope. Tous ces attributs sont optionnels et si l'attribut var est présent, le type de la variable créée est java.lang.Integer. La variable contient le nombre de lignes modifiées par la requête. Cette action peut contenir des actions <sql:param>.

Utilisation de la JSTL

Nous allons nous servir de notre application Web messagerie afin de placer certaines actions de la JSTL. Ainsi, nous éliminerons complètement tout code Java écrit directement dans les pages JSP.

<fmt:formatDate>

Toutefois, dans un premier temps, nous allons revenir sur le pieds de page afin de mettre en oeuvre l'action prévue pour le formatage de la date et ainsi remplacer l'action personnalisée que nous avons construite.

pieds.jspf

```
1 <%@taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt" %>
2
3         <br><hr>
4         <jsp:useBean id="dateActuelle" scope="page" class="java.util.Date" />
5         <h4 align="right"><fmt:formatDate value="${dateActuelle}" dateStyle="full"/></h4>
6       </font>
7     </body>
8 </html>
9
```

La balise <fmt:formatDate> ne sert que de formatage. Il est nécessaire de créer la date, ce que nous faisons au travers d'un bean dateActuelle.

Finalement, l'investissement qui consiste à créer notre propre balise personnalisée intégrant à la fois la création ainsi que son formatage vaut le coup. Effectivement, nous voyons ici que nous devons écrire deux lignes (5 et 6) pour afficher une simple date dans le pied de page. La date personnalisée que nous avons mis en oeuvre peut être utilisée pour d'autres applications Web. Nous l'avons conçue pour cela. Elle est beaucoup plus facile à utiliser puisque sa syntaxe est très concise.

Le fragment de page <navigation.jspf>

Nous allons profiter de la JSTL pour restructurer l'en-tête de notre site. En effet, le fragment de page associé possède un bon mélange entre le codage Java et le codage HTML proprement dit. Les actions de la JSTL vont nous permettre de construire un nouveau fragment de page beaucoup plus agréable à lire.

Je rappelle le fragment de page <navigation.jspf> tel que nous l'avons avant la transformation :

navigation.jspf

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">

<%@ page errorPage = "/WEB-INF/erreur.jsp" import="bd.*" %>

<%! int idPersonne = 1; %>

<%
    String identité = "A tout le monde";
    Personne opérateur = (Personne) session.getAttribute("utilisateur");
    if (opérateur != null) {
        idPersonne = opérateur.identificateur();
        identité = opérateur.getPrénom()+" "+opérateur.getNom();
    }
    pageContext.setAttribute("personne", identité);
%>

<html>
  <head><title>Messages</title></head>
  <body bgcolor="#${initParam.couleurFond}">
    <font face="Arial">
        <h2 align="center">Messages</h2>
        <hr>
        <table bgcolor="1" cellpadding="3" cellspacing="2" width="90%" align="center">
            <tr bgcolor="#FF9900">
                <th align="left"><a href="bienvenue">Sujets</th>
                <th align="right">
                    <% if (idPersonne == 1) { %>
                        <a href="utilisateur?authentification=personnel">Identification</a>
                        <a href="utilisateur?authentification=nouveau">Inscription</a>
                    <% }
                       else { %>
                         <a href="#">Nouveau</a>
                         <a href="#">Modifier</a>
                         <a href="#">Enlever</a>
                    <% } %>
                </th>
            </tr>
            <tr>
                <th align="left">${personne}</th>
                <th align="right">${pageContext.request.remoteHost}</th>
            </tr>
        </table>

```

Vous remarquez qu'effectivement nous avons un bon mélange entre le codage HTML et les écritures Java : déclarations , scriptlets, etc. Voici maintenant ce que nous obtenons en utilisant la JSTL :

navigation.jspf

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">

<%@ page errorPage = "/WEB-INF/erreur.jsp" import="bd.*" %>
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>

 <c:choose>
      <c:when test="${empty utilisateur}">
            <c:set var="idPersonne" value="1" />
            <c:set var="identité" value="A tout le monde" />
      </c:when>
      <c:otherwise>
            <c:set var="idPersonne" value="${utilisateur.identificateur}" />
            <c:set var="identité" value="${utilisateur.prénom} ${utilisateur.nom}" />
      </c:otherwise>
 </c:choose>

<html>
  <head><title>Messages</title></head>
  <body bgcolor="#${initParam.couleurFond}">
    <font face="Arial">
        <h2 align="center">Messages</h2>
        <hr>
        <table bgcolor="1" cellpadding="3" cellspacing="2" width="90%" align="center">
            <tr bgcolor="#FF9900">
                <th align="left"><a href="bienvenue">Sujets</th>
                <th align="right">
                    <c:choose>
                       <c:when test="${idPersonne == 1}" >
                           <a href="utilisateur?authentification=personnel">Identification</a>
                           <a href="utilisateur?authentification=nouveau">Inscription</a>
                       </c:when>
                       <c:otherwise>
                           <a href="#">Nouveau</a>
                           <a href="#">Modifier</a>
                           <a href="#">Enlever</a>
                       </c:otherwise>
                    </c:choose>
                </th>
            </tr>
            <tr>
                <th align="left">${identité}</th>
                <th align="right">${pageContext.request.remoteHost}</th>
            </tr>
        </table>

```

La page est effectivement plus facile à lire. La remarque que nous pouvons faire, c'est que l'action <c:if> ne possède pas de <c:else>. Nous devons utiliser la combinaison <c:choose>, <c:when> et <c:otherwise>.

Validation de l'utilisateur

De même nous allons revoir la validation de l'utilisateur puisque cette page possède également en fin de partie, à partir de la ligne 28, un petit peu de codage Java :

validerutilisateur.jsp

```
<%@ page errorPage = "/WEB-INF/erreur.jsp" import="bd.*" %>
<%! int idPersonne = 1; %>
<%! String identité = "Confirmation de vos références"; %>
<%@ include file = "/WEB-INF/jspf/navigation.jspf" %>

<h3 align="center">Confirmation de votre demande d"inscription</h3>

<jsp:useBean id="utilisateur" class="bd.Personne" scope="session">
    <jsp:setProperty name="utilisateur" property="*" />

    <p><table border="1" cellpadding="3" cellspacing="2" width="90%" align="center">
        <tr>
            <td bgcolor="#FF9900" width="100"><b>Nom</b></td>
            <td><jsp:getProperty name="utilisateur" property="nom" /></td>
        </tr>
        <tr>
            <td bgcolor="#FF9900" width="100"><b>Prénom</b></td>
            <td><jsp:getProperty name="utilisateur" property="prénom" /></td>
        </tr>
        <tr>
            <td bgcolor="#FF9900" width="100"><b>Mot de passe</b></td>
            <td><jsp:getProperty name="utilisateur" property="motDePasse" /></td>
        </tr>
    </table></p>
</jsp:useBean>

<h3 align="center">
<% if (!utilisateur.enregistrer()) {  %>
     <font color="red">ATTENTION : Utilisateur déja enregistré</font>
<%
        session.removeAttribute("utilisateur");
    } else {
%>
     <font color="green">Nouvel utilisateur enregistré</font>
<% } %>
</h3>

<%@ include file = "/WEB-INF/jspf/pieds.jspf" %>
```

Voici ce que nous pouvons modifier pour éliminer ce codage Java directement sur le page JSP :

validerutilisateur.jsp

```
<%@ include file = "/WEB-INF/jspf/navigation.jspf" %>

<h3 align="center">Confirmation de votre demande d'inscription</h3>

<jsp:useBean id="utilisateur" class="bd.Personne" scope="session">
    <jsp:setProperty name="utilisateur" property="*" />

    <p><table border="1" cellpadding="3" cellspacing="2" width="90%" align="center">
        <tr>
            <td bgcolor="#FF9900" width="100"><b>Nom</b></td>
            <td>${utilisateur.nom}</td>
        </tr>
        <tr>
            <td bgcolor="#FF9900" width="100"><b>Prénom</b></td>
            <td>${utilisateur.prénom}</td>
        </tr>
        <tr>
            <td bgcolor="#FF9900" width="100"><b>Mot de passe</b></td>
            <td>${utilisateur.motDePasse}</td>
        </tr>
    </table></p>
</jsp:useBean>

<h3 align="center">
   <c:choose>
     <c:when test="${utilisateur.enregistrer}">
           <font color="green">Nouvel utilisateur enregistré</font>
     </c:when>
     <c:otherwise>
           <font color="red">ATTENTION : Utilisateur déja enregistré</font>
           <c:remove scope="session" var="utilisateur" />
     </c:otherwise>
   </c:choose>
</h3>

<%@ include file = "/WEB-INF/jspf/pieds.jspf" %>
```

C'est tout de même plus facile à lire !
.

Requête à la base de données

Pour terminer, nous allons utiliser les actions JSTL pour établir une communication avec la base de données. Nous allons donc reprendre la page <bienvenue.jsp> et remplacer les actions personnalisées par ceux de la JSTL. Revoyons donc la page que nous avions déjà développée :

bienvenue.jsp

```
<%@ include file = "/WEB-INF/jspf/navigation.jspf" %>
<%@taglib uri="/util" prefix="u" %>

<u:tableau table="message" identificateur="idPersonne" valeur="${idPersonne}" couleur="#FF6600" largeur="90%">
   <u:Colonne titre="sujet">Sujet</u:Colonne>
   <u:Colonne titre="texte">Message correspondant</u:Colonne>
</u:tableau>

</font>

<%@ include file = "/WEB-INF/jspf/pieds.jspf" %>
```

Nous pouvons remarquer que nous pouvons difficilement faire plus concis, puisque toute la partie codage du tableau HTML se trouve à l'intérieur du gestionnaire de balise Tableau. Toutefois, si nous devons implémenter rapidement un tableau représentant une table de la base de données, voici comment nous pouvons procéder :

bienvenue.jsp

```
<%@ include file = "/WEB-INF/jspf/navigation.jspf" %>
<%@taglib uri="http://java.sun.com/jsp/jstl/sql" prefix="sql" %>

<sql:query var="messages"
           dataSource="jdbc:mysql://127.0.0.1/messagerie,com.mysql.jdbc.Driver,root,manu"
           sql="SELECT * FROM message" />

<table border="1" cellpadding="3" cellspacing="2" width="90%" align="center">
  <tr bgcolor="#FF6600">
      <th>Sujet</th>
      <th>Messages</th>
  </tr>
  <c:forEach var="message" items="${messages.rows}">
      <tr>
         <td><c:out value="${message.sujet}"/></td>
         <td><c:out value="${message.texte}"/></td>
      </tr>
  </c:forEach>
</table>

</font>

<%@ include file = "/WEB-INF/jspf/pieds.jspf" %>
```

Il est vrai que cette page est largement plus longue que la précédente. Toutefois, le code demeure très clair et effectivement il reste avantageux de connaître la JSTL. Malgré tout, comme nous le voyons ici, je pense que nous ne devons pas hésiter à fabriquer nos propres actions personnalisées. L'investissement vaut largement le coup, surtout si nous pouvons déployer ces actions personnalisées pour d'autres applications Web.

Au moyen de cette étude, nous nous sommes appliquer à construire des pages JSP beaucoup plus facile à lire. Elle sont, la plupart du temps très concise, et le webmaster passera donc moins de temps à les concevoir. En conséquence, ces pages seront également plus faciles à maintenir.