Présentation des Expressions Languages.

par F. Martini (adiGuba)

Date de publication : 15/01/2005

Dernière mise à jour : 16/01/2005

Ce tutoriel décrit le fonctionnement des Expressions Languages (EL) de JSP 2.0. Ce tutoriel est également disponible en version PDF : el.pdf.

Remerciement

Je tiens à remercier Ukyuu pour avoir pris le temps de relire ce tutoriel...

1 - Qu'est-ce que les Expressions Languages ?

Les **Expressions Languages** (**EL**) permettent de manipuler les données au sein d'une page JSP (ou d'un fichier *.tag) plus simplement qu'avec les scriptlets Java.

Une EL permet d'accéder simplement aux beans des différents scopes de l'application web (**page**, **request**, **session** et **application**). Utilisé conjointement avec des librairies de tags, elles permettent de se passer totalement des scriptlets.

Une expression EL est de la forme suivante :

```
${ expression }
```

La chaîne **expression** correspond à l'expression à interpréter. Une expression peut être composée de plusieurs termes séparés par des opérateurs (Voir "Les opérateurs") :

```
${ terme1 opérateur terme2 }
${ opérateur-unaire terme }
${ terme1 opérateur terme2 opérateur terme3 [opérateur terme*]... }
etc.
```

Les différents termes peuvent être :

- Un type primaire (Voir "Les types primaires").
- Un objet implicite (Voir "Les objets implicites").
- Un attribut d'un scope de l'application (Voir "Les attributs des différents scopes de l'application").
- Une fonction EL (Voir "Les fonctions").

Les **EL** permettent également d'accéder simplement aux propriétés des objets (Voir "Accès aux propriétés des objets") :

```
${ object.property }
${ object["index property"] }
${ object[index] }
${ object[0] }
ect...
```

1.1 - Comment utiliser les EL ?

Afin d'utiliser les **EL**, une application web J2EE 1.4 (Servlet 2.4 / JSP 2.0) est nécessaire. Les **EL** peuvent alors être utilisées dans n'importe quel page JSP ou fichier *.tag :

- Dans les attributs des tags JSP.
- Dans du texte simple de la page JSP.

Par exemple :

```
${monBean}
<prefix:actionTag id="newBean" param="${monBean}">
        ${newBean}
</prefix:actionTag>
```

*Il est possible d'indiquer au conteneur JSP 2.0 de ne pas interpréter les **EL** d'une page ou d'un tag. Il suffit pour cela d'utiliser l'attribut **isELIgnored** de la directive **page** ou **tag** :*

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

Il est également possible de ne pas interpréter une **EL** en particulier (ou une chaîne commençant par "**${**"...) en la protégeant avec un anti-slash :

```
\${ ceci ne sera pas interprété comme une EL }
```

*Il est également possible d'utiliser les **EL** avec **JSTL 1.0** (Java Standard Tag Library) avec un serveur d'application **J2EE 1.3**.*

*En effet, le groupe de travail chargé de définir la **JSTL** (JSR-52) a travaillé en étroite collaboration avec le groupe de travail chargé de formaliser les spécifications des **JSP 2.0** (JSR-152).*

*La **JSTL** est fortement basé sur l'utilisation des **EL**. Toutefois, les spécifications de la **JSTL** ont été finalisées plus d'un an avant celle des **JSP 2.0**. C'est pourquoi la version 1.0 des **JSTL** a donc intégré une gestion partielle des **EL**.*

*Mais contrairement aux **JSP 2.0**, ce n'est pas le conteneur JSP qui gère les **EL**, mais les tags eux-mêmes. Donc il est impossible dans ce cas d'utiliser les **EL** ailleurs que pour les attributs des tags de la JSTL...*

*Les **JSTL 1.0** propose ainsi le tag **<c:out/>** pour afficher indirectement le contenu d'une **EL** sur une page JSP :*

Exemple d'utilisation de c:out

```
<c:out value="${expression}" />
```

1.2 - Les types primaires

Les principaux types primaires de Java peuvent être utilisé dans les expressions. Cependant, tous les éléments des **EL** restent des objets, c'est pourquoi ils sont convertis en objet du type de la classe **wrapper** correspondante (classe représentant un type primaire), comme le montre le tableau suivant :

Termes	Description	Type
null	La valeur **null**.	-
123	Une valeur entière.	**java.lang.Long**
123.00	Une valeur réelle.	**java.lang.Double**
"chaîne" ou **'chaîne'**	Une chaîne de caractère.	**java.lang.String**
true ou **false**	Un booléen.	**java.lang.Boolean**

Contrairement aux langage Java, les chaînes de caractères peuvent être représenté à la fois avec des quottes simples ou doubles, ceci afin de simplifier leurs utilisations au sein des attributs de tag...

1.3 - Les objets implicites

En plus des types primaires, un certains nombres d'objets implicites permettent d'accéder aux différents composants d'une page JSP :

- **pageContext** : Accès à l'objet **PageContext** de la page JSP.
- **pageScope** : Map permettant d'accéder aux différents attributs du scope **'page'**.
- **requestScope** : Map permettant d'accéder aux différents attributs du scope **'request'**.
- **sessionScope** : Map permettant d'accéder aux différents attributs du scope **'session'**.
- **applicationScope** : Map permettant d'accéder aux différents attributs du scope **'application'**.
- **param** : Map permettant d'accéder aux paramètres de la requête HTTP sous forme de **String**.
- **paramValues** : Map permettant d'accéder aux paramètres de la requête HTTP sous forme de **tableau de String**.
- **header** : Map permettant d'accéder aux valeurs du Header HTTP sous forme de **String**.
- **headerValues** : Map permettant d'accéder aux valeurs du Header HTTP sous forme de **tableau de String**.
- **cookie** : Map permettant d'accéder aux différents Cookies.
- **initParam** : Map permettant d'accéder aux **init-param**s du web.xml.

Quelques exemples d'utilisations :

${ pageContext.response.contentType } affiche le content type de la réponse.

${ pageScope["name"] } affiche l'attribut "**name**" du scope page.

${ param["page"] } affiche la valeur du paramètre "**page**".

${ header["user-agent"] } affiche l'header "**user-agent**" envoyé par le navigateur.

etc.

Voir la section "Accès aux propriétés des objets" pour plus de détail sur l'accès aux données...

1.4 - Les attributs des différents scopes de l'application

Lors de l'évaluation d'un terme, si celui ci n'est ni un type primaire, ni un objet implicite, le conteneur JSP recherchera alors un attribut du même nom dans les différents scopes de l'application en utilisant la méthode **findAttribute()** du **PageContext** de la page JSP.

Ainsi, l'expression suivante :

EL
```
${ name }
```

Correspond au code suivant :

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

L'attribut "name" sera donc recherché successivement dans les différents scopes (dans l'ordre : **page**, **request**,

session, **application**).

*Il est conseillé de n'utiliser cette forme seulement pour accéder aux objets du scope **page**, afin d'éviter d'éventuel conflit...*

*Pour les autres scopes, il est préférable d'utiliser les objets implicites **requestScope["name"]**, **sessionScope["name"]**, et **applicationScope["name"]**...*

1.5 - Les opérateurs

Il est également possible d'utiliser des opérateurs dans une **EL**. Ils sont particulièrement utiles lorsqu'ils sont utilisés avec les tags conditionnels de la **JSTL** (ou d'une autre librairie de tag).

Il s'agit des mêmes opérateurs que ceux du langage Java, mis à part que certains possèdent un équivalent textuelle afin d'éviter des conflits lorsque les **EL** sont utilisées dans une balise, ou que l'on souhaite que le fichier source JSP soit un fichier XML valide...

Les opérateurs s'opèrent sur deux termes et prennent la forme suivante :

terme1 opérateur terme2 (Mis à part quelques exception -- voir plus bas).

Opérateurs arithmétiques :	
+	Addition
-	Soustraction
*****	Multiplication
/ ou **div**	Division
% ou **mod**	Modulo

Opérateurs relationnels :	
== ou **eq**	Egalité
!= ou **ne**	Inégalité
< ou **lt**	Plus petit que
> ou **gt**	Plus grand que
<= ou **le**	Plus petit ou égal
>= ou **ge**	Plus grand ou égal

Opérateurs logiques :	
&& ou **and**	ET logique (**true** si les deux opérandes valent **true**, **false** sinon)
|| ou **ot**	OU logique (**true** si au moins une des deux opérandes vaut **true**, **false** sinon)
! ou **not** [*]	NON logique (**true** si l'opérande vaut **false**, et inversement)

Autres :	
empty [*]	**true** si l'opérande est **null**, une chaîne vide, un tableau vide, une Map vide ou une List vide. **false** sinon.
( ) [*]	Modifie l'ordre des opérateurs.
? :	Opérateur conditionnel en ligne, format particulier : *condition* **?** *valeur_si_true* **:** *valeur_si_false*

Les **opérateurs arithmétiques** ne peuvent être utilisés que sur des données numériques.

Les **opérateurs logiques** ne s'appliquent que sur des booléens.

Les **opérateurs relationnels** d'égalité et d'inégalité utilisent la méthode **equals()** d'**Object**. Il est donc conseillé de la redéfinir si on se sert de ces opérateurs.

Les **opérateurs relationnels** de comparaison ne s'appliquent que si l'interface **Comparable** est implémentée La méthode **compareTo()** est alors utilisée.

[*] *Les opérateurs* ***!****,* ***not*** *et* ***empty*** *sont des opérateurs unaires, c'est à dire qu'il sont de la forme suivante :*

opérateur terme

L'opérateur ***()*** *permet de changer l'ordre des opérateurs. L'ordre d'exécution des opérateurs est le même que dans le langage Java, c'est à dire que l'expression* ***${ 10 + 2 * 2 }*** *donnera 14 tandis que* ***${ (10 + 2) * 2 }*** *donnera 24...*

1.6 - Accès aux propriétés des objets

Trois catégories d'**objets** sont définit par les **EL** :

- Les **beans mappés** (objet de type **Map**)
- Les **beans indexées** (**tableau** Java, ou objet de type **List**)
- Les **beans** standards (tout autres objets).

Selon le type de bean, les règles d'accès à ses propriétés divergent malgré une syntaxe similaire. Ces catégories sont également valables pour les propriétés des beans.

L'accès aux beans se fait par réflexivité, c'est à dire qu'il n'y a pas besoin de connaître le type de l'objet pour accéder à ses propriétés, le conteneur JSP recherche dynamiquement la propriété.

1.3.1 - Les propriétés des beans standards

Les méthodes **accesseurs** sont utilisées par réflexivité afin d'accéder aux propriétés d'un bean. C'est à dire que pour accéder à la propriété **name**, la méthode **getName()** est recherché puis appelé.

Il y a deux manière d'accéder aux propriétés d'un bean : en utilisant l'opérateur point (**.**) ou l'opérateur crochet (**[...]**).

Ainsi, pour accéder à la propriété **name** de notre bean, on peut utiliser les syntaxes suivantes :

Opérateur 'point'
```
${ bean.name }
```

Opérateur 'crochet'
```
${ bean["name"] }
ou
${ bean['name'] }
```

Dans les deux cas le résultat sera le même et est équivalent au code suivant :

Code Java
```
<%@ page import="package.MonBean" %>
<%
        MonBean bean = (MonBean) pageContext.findAttribute ("bean");
        if (bean != null)
                out.print ( bean.getName() );
%>
```

Les **EL** apportent trois avantages :

- Une meilleure **lisibilité** : le code se limite quasiment au nom du bean et de sa propriété.
- Pas de **cast**, et de ce fait pas d'import (on accède aux propriétés par réflexion).
- Gestion des valeurs **null** (Voir la section "Gestion des exceptions).


*Quelque soit l'opérateur utilisé, une exception sera lancé si le bean ne comporte pas d'**accesseur** pour la propriété.*

Par rapport à l'opérateur 'point', l'opérateur 'crochet' utilise une chaîne de caractère pour déterminer le nom de la propriété. Cette chaîne peut très bien être le résultat de l'évaluation d'un autre bean, par exemple, si **config.propertyName** comporte le nom de la propriété a affiché :

```
${ bean[ config.propertyName ] }
```


*En réalité l'opérateur 'crochet' accepte n'importe quel objet. La méthode **toString()** est utilisé afin de récupéré la chaîne de caractères correspondant au nom de la propriété.*

Enfin, les deux opérateurs peuvent bien sûr être utilisés conjointement au sein d'une même expression. Par exemple, pour accéder à la propriété **city** de la propriété **address** du bean **person**, on peut utiliser les différentes formes suivantes :

```
${ person.address.city }
${ person.['address'].['city'] }
${ person.["address"].["city"] }
${ person.address.['city'] }
${ person.["address"].city }
etc.
```


L'opérateur 'crochet' est également utilisé pour accéder aux beans indexés et mappés, il est donc fortement conseillé de lui préférer l'opérateur 'point' lorsque c'est possible afin d'éviter les confusions.

1.3.2 - Les propriétés indexées (List et tableau)

Les propriétés indexées permettent d'accéder aux différents éléments d'un tableau ou d'une **List**.

On peut accéder aux différents éléments en utilisant l'opérateur 'crochet' avec un index, par exemple :

```
${ list[0] }
${ list[1] }
${ list["2"] }
${ list["3"] }
${ list[ config.value ] }
etc.
```

On accède alors aux différentes valeurs via la méthode **get(int)** de l'interface **List** (ou un accès au nième éléments du tableau).

Si l'index n'est pas un nombre entier, il sera automatiquement convertit (ce qui risque de provoquer une exception : voir la section "Gestion des exceptions").

*L'opérateur 'point' ne peut pas être utilisé sur un tableau ou une **List**...*

1.3.3 - Les propriétés mappés (Map)

De la même manière que pour les propriétés indexées, on utilise l'opérateur 'crochet' afin d'accéder aux différentes valeurs d'une **Map**, par exemple :

```
${ map["clef1"] }
${ map['clef2'] }
${ map[ config.key ] }
etc.
```

Le conteneur JSP utilisera la méthode **get(Object)** de l'interface **Map** en utilisant l'objet entre les crochets comme clef.

*L'opérateur 'point' peut également être utilisé sur un bean mappé, mais il a le même comportement que l'opérateur 'crochet'. C'est à dire qu'il n'utilise pas l'**accesseur** de la propriété, mais il utilise le nom de la propriété en tant que clef pour la méthode **get()**.*

*Cela reste toutefois déconseillé afin de bien distinguer les **Map** des autres beans...*

1.7 - Les fonctions

Il est possible d'associer un nom de fonction à une méthode statique publique de n'importe quelle classe Java. Il faut pour cela définir le lien entre le nom de la fonction **EL** et celui de la méthode statique (qui peuvent être différent) dans un descripteur de taglib.

En effet, les fonctions **EL** sont associées aux librairies de tag. Pour plus de détail sur la déclaration de fonction **EL** dans une taglib, veuillez consulter le tutoriel consacré aux taglib (Section **3.5. Les fonctions EL**) :

http://adiguba.developpez.com/tutoriels/j2ee/jsp/taglib/#L3.5

*L'utilisation de fonction **EL** nécessite un serveur d'application J2EE 1.4 au minimum.*

Ainsi, une fonction **EL** nécessite l'utilisation de la directive **<%@ taglib %>** et du préfixe de la librairie de tag auquel elle appartient. Une fonction **EL** a donc la forme suivante :

${ prefix:nomFonction ([parametres...]) }

Par exemple, la **JSTL 1.1** comporte un librairie de tag dédiée à diverses fonctions de traitement de chaîne de caractères.

Ainsi la fonction **escapeXml()** permet de protéger les caractères qui peuvent être interprétés par le langage XML (et donc HTML) :

Utilisation de la fonction escapeXml
```
<%@ taglib uri="http://java.sun.com/jsp/jstl/functions" prefix="fn" %>
${ fn:escapeXml("<balise>") }
```

Les caractères "**<**" et "**>**" seront ainsi converti respectivement en "**<**" et "**>**"...

Il est bien entendu possible d'utiliser n'importe quel terme comme paramètre de la fonction :

Exemples
```
<%@ taglib uri="http://java.sun.com/jsp/jstl/functions" prefix="fn" %>
${ fn:escapeXml( param['text'] ) }
${ fn:escapeXml( bean.data.message.text ) }
${ fn:escapeXml( $fn:trim(param['text']) ) }
etc.
```

1.8 - Gestion des exceptions

Les **EL** gèrent un certains nombres d'exceptions afin de ne pas avoir à les traiter dans les JSP.

NullPointeurException (et les valeurs **null**) :

Les différentes propriétés d'un élément ne sont pas forcément renseigner et peuvent très bien être **null**. Par exemple, dans l'expression **${ sessionScope['data'].information.date }**, plusieurs éléments peuvent être **null**. En effet, l'attribut 'data' peut ne pas être présent dans la **session**, ou alors sa méthode **getInformation()** peut renvoyer **null**...

Dans tous les cas, l'expression prendra la valeur **null** mais aucune exception ne sera lancée.

De plus, lors de l'affichage dans la page JSP, toutes les valeurs **null** sont remplacées par des chaînes vides, afin de ne pas afficher le texte **null** à l'utilisateur...

ArrayOutOfBoundException :

De la même manière, l'accès à un index incorrect d'un élément d'un tableau ou d'une **List** ne provoquera pas d'exception mais renverra une valeur **null**.

Enfin, un ensemble de règles approfondies permet au conteneur JSP d'effectuer la plupart des conversions de type nécessaire à une application web. Ainsi, le type exact des données n'est plus un problème...

La logique de ces mécanismes se place bien évidemment du coté des utilisateurs de l'application : Soit la donnée à afficher est valide alors on l'affiche, soit elle n'est pas présente et on n'affiche rien...

Cela permet de se passer d'effectuer plusieurs vérifications au sein des JSP avant d'afficher des données...

Toutefois, certaines exceptions peuvent toujours être lancées, mais elles ne concernent pas directement les données mais l'expression **EL**. Par exemple, lorsque l'expression est mal formé, que l'on tente d'accéder à une propriété sans accesseur publique, ou que l'on accède à une propriété indexée avec un index qui ne correspond pas à un nombre entier...

Une **ELException** est alors lancé avec un texte décrivant l'erreur.

2 - Exemples d'utilisations

Cette partie décrit quelques exemples concrets d'utilisations des **EL**. Pour chaque exemple, une version avec des **scriptlets** est également présentée afin de bien se rendre compte de l'intérêt des **EL**...

Des tags de la **JSTL** ont également été utilisés dans les exemples avec les **EL**.

*Il existe des frameworks qui permettant d'obtenir un résultat proche (notamment **Struts** ou **Spring**). Toutefois, l'utilisation du couple **EL/JSTL** permet de gérer un grand nombre de cas typique de n'importe quelle application web...*

2.1 - Remplir les champs d'un formulaire

Imaginons un site avec authentification (un peu comme les forums de developpez.com). Lorsque un utilisateur se connecte, son profil est chargé en session, et une page lui permet de modifier les informations le concernant.

On va donc générer le formulaire permettant de modifier les informations du profil. Le profil étant un objet stocké en **session** correspondant à la classe suivante :

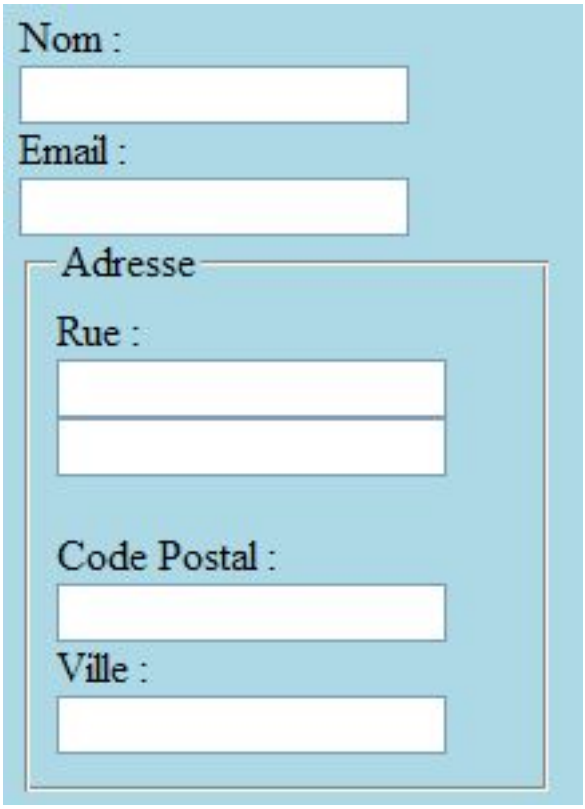
Profil
```
public class Profil implements Serializable {
        private String name;
        private String email;
        private Address address;

        public Address getAddress() {
                return address;
        }
        public String getEmail() {
                return email;
        }
        public String getName() {
                return name;
        }
}
```

Avec **Address** correspondant à la classe suivante :

Address
```
public class Address implements Serializable {
        private String line1;
        private String line2;
        private String zipCode;
        private String city;

        public String getLine1() {
                return line1;
        }

        public String getLine2() {
                return line2;
        }

        public String getZipCode() {
                return zipCode;
        }
        public String getCity() {
                return city;
        }
}
```

Le code de ces classes a volontairement été réduit afin de se limiter aux informations utilisées lors de l'affichage...

Ainsi, on utilisera le formulaire basique suivant pour permettre à l'utilisateur de modifier son profil ;



Formulaire obtenu simplement avec le code HTML suivant :

Formulaire HTML
```
<form action="/monAction" method="post">

        Nom :<br/>
        <input type="text" name="name" value=""/><br/>
        Email :<br/>
        <input type="text" name="email" value=""/><br/>

        <fieldset>
                <legend>Adresse</legend>
                Rue :<br/>
                <input type="text" name="line1" value=""/><br/>
                <input type="text" name="line2" value=""/><br/>
                <br/>
                Code Postal :<br/>
                <input type="text" name="zipCode" value=""/><br/>
                Ville :<br/>
                <input type="text" name="ville" value=""/><br/>
        </fieldset>

</form>
```

Il reste donc à renseigner ce formulaire afin d'éviter à l'utilisateur de devoir ressaisir toutes les informations à chaque modification...

2.1.1 - Utilisation des scriptlets

Dans un premier temps on se contente donc de récupérer le profil et de renseigner les différents champs **value** du formulaire HTML :

Exemple scriptlet

```
<%@ page import="packageNames.*" %>
<% Profil p = (Profil) session.getAttribute ( "profil" ); %>
<form action="/monAction" method="post">

        Nom :<br/>
        <input type="text" name="name" value="<%=p.getName()%>"/><br/>
        Email :<br/>
        <input type="text" name="email" value="<%=p.getEmail()%>"/><br/>

        <fieldset>
                <legend>Adresse</legend>
                Rue :<br/>
                <input type="text" name="line1" value="<%=p.getAddress().getLine1()%>"/><br/>
                <input type="text" name="line2" value="<%=p.getAddress().getLine2()%>"/><br/>
                <br/>
                Code Postal :<br/>
                <input type="text" name="zipCode" value="<%=p.getAddress().getZipCode()%>"/><br/>
                Ville :<br/>
                <input type="text" name="ville" value="<%=p.getAddress().getCity()%>"/><br/>
        </fieldset>

</form>
```

Problèmes : on ne gère pas les cas où le profil est absent ou qu'il ne possède pas d'adresse (**getAddress()** retourne **null**), et on risque donc d'obtenir des exceptions... De plus, si certains champs sont absent (comme la seconde ligne de l'adresse), la chaîne **null** est affichée.

On peut résoudre ces problèmes simplement en effectuant quelques tests et en utilisant des variables de scripts. Le code n'est pas bien méchant mais dévient quand même assez important :

Exemple scriptlet (2)

```
<%@ page import="packageNames.*" %>
<%
        Profil p = (Profil) session.getAttribute ( "profil" );
        String name = null;
        String email = null;
        String line1 = null;
        String line2 = null;
        String zipcode = null;
        String city = null;
        if ( p!=null ) {
                name = p.getName();
                email = p.getEmail();
                Address address = p.getAddress();
                if ( address!=null ) {
                        line1 = address.getLine1();
                        line2 = address.getLine2();
                        zipcode = address.getZipCode();
                        city = address.getCity();
                }
        }
        if ( name==null ) name = "";
        if ( email==null ) email = "";
        if ( line1==null ) line1 = "";
        if ( line2==null ) line2 = "";
        if ( zipcode==null ) zipcode = "";
        if ( city==null ) city = "";

%>
<form action="/monAction" method="post">

        Nom :<br/>
        <input type="text" name="name" value="<%=name%>"/><br/>
        Email :<br/>
        <input type="text" name="email" value="<%=email%>"/><br/>
```

Exemple scriptlet (2)

```
            <fieldset>
                    <legend>Adresse</legend>
                    Rue :<br/>
                    <input type="text" name="line1" value="<%=line1%>"/><br/>
                    <input type="text" name="line2" value="<%=line2%>"/><br/>
                    <br/>
                    Code Postal :<br/>
                    <input type="text" name="zipCode" value="<%=zipcode%>"/><br/>
                    Ville :<br/>
                    <input type="text" name="ville" value="<%=city%>"/><br/>
            </fieldset>

    </form>
```

On se retrouve donc avec un code assez conséquent pour afficher seulement 6 valeurs : notre page JSP (qui ne devrait servir qu'à la présentation) est ainsi 'polluer' de plusieurs lignes codes afin de vérifier les données...

Les vérifications occupent à peu près la moitié du code de la page...

2.1.2 - Utilisation des EL

Nous allons désormais utiliser les **EL** et l'objet implicite **sessionScope** afin d'accéder directement aux attributs de la session.

Notre formulaire devient simplement :

Code EL

```
    <form action="/monAction" method="post">

            Nom :<br/>
            <input type="text" name="name" value="${sessionScope['profil'].name}"/><br/>
            Email :<br/>
            <input type="text" name="email" value="${sessionScope['profil'].email}"/><br/>

            <fieldset>
                    <legend>Adresse</legend>
                    Rue :<br/>
                    <input type="text" name="line1"
    value="${sessionScope['profil'].address.line1}"/><br/>
                    <input type="text" name="line2"
    value="${sessionScope['profil'].address.line2}"/><br/>
                    <br/>
                    Code Postal :<br/>
                    <input type="text" name="zipCode"
    value="${sessionScope['profil'].address.zipCode}"/><br/>
                    Ville :<br/>
                    <input type="text" name="ville"
    value="${sessionScope['profil'].address.city}"/><br/>
            </fieldset>

    </form>
```

Toutes les vérifications sont effectuées par les **EL** et les valeurs **null** sont remplacées par des chaînes vides.

Cet exemple nous offre donc un code correct sans avoir a effectué de pénibles vérifications...

On peut toutefois utiliser le tag **<c:set/>** de la **JSTL** qui permet de placer un objet dans un autre scope afin d'éviter les répétitions de **sessionScope['profil']** :

Code EL
```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="d" %>
<d:set var="p" value="${sessionScope['profil']}"/>
<form action="/monAction" method="post">

        Nom :<br/>
        <input type="text" name="name" value="${p.name}"/><br/>
        Email :<br/>
        <input type="text" name="email" value="${p.email}"/><br/>

        <fieldset>
                <legend>Adresse</legend>
                Rue :<br/>
                <input type="text" name="line1" value="${p.address.line1}"/><br/>
                <input type="text" name="line2" value="${p.address.line2}"/><br/>
                <br/>
                Code Postal :<br/>
                <input type="text" name="zipCode" value="${p.address.zipCode}"/><br/>
                Ville :<br/>
                <input type="text" name="ville" value="${p.address.city}"/><br/>
        </fieldset>

</form>
```

*Il est également possible d'utiliser **${profil.name}** à la place de **${sessionScope['profil'].name}**, mais il faut alors être certain qu'il n'existe pas d'attribut du même nom dans les scopes **page** et **request**...*

*L'utilisation du tag **<c:set/>** permet d'éviter ce type de problème.*

2.2 - Afficher le contenu d'une collection

Les applications web affichent généralement des données en provenance de sources diverses (fichiers, services web, base de données, etc.). Généralement, afin de bien séparer le code métier de l'affichage, les données sont stockées dans une collection qui est passé à la page JSP via le **request**.

Nous allons seulement donc écrire la page JSP qui affichera les données. Nous prendrons en exemple l'affichage d'une liste de livre.

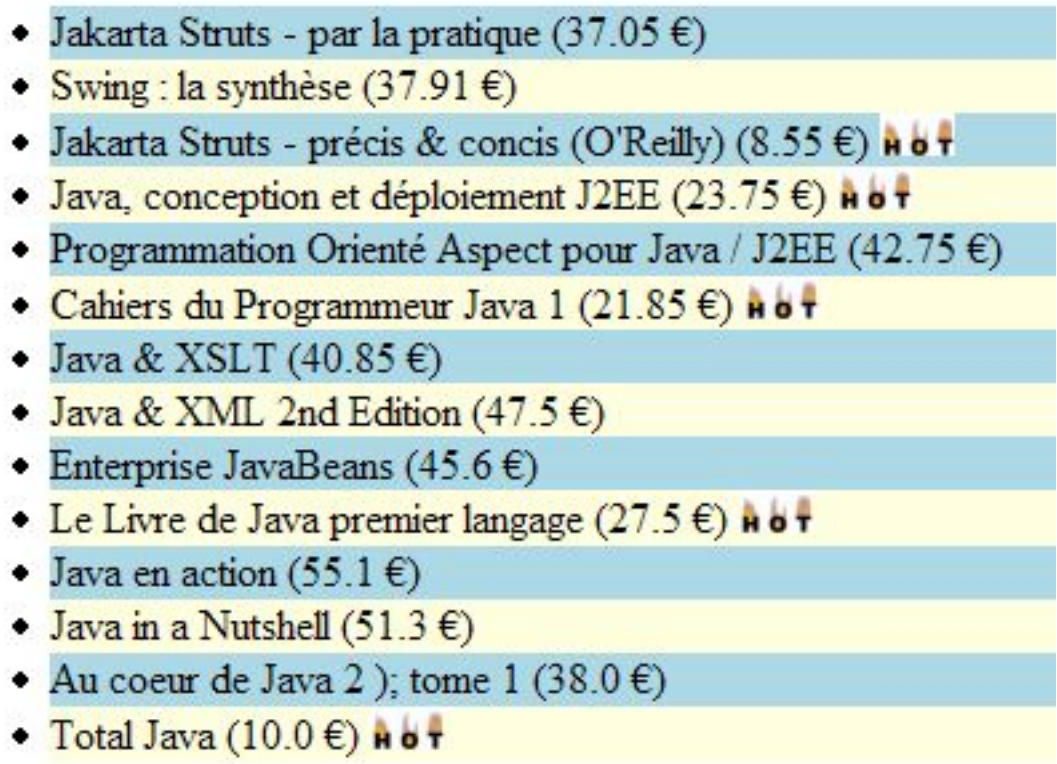
Lorsque notre JSP sera exécutée, le scope **request** possèdera donc un attribut "**books-list**" contenant une **List** d'objet de type **Book**, définit ci dessous :

Book
```
public class Book {
        private String name;
        private double price;

        public String getName() {
                return name;
        }

        public double getPrice() {
                return price;
        }
}
```

On affichera dans la liste le nom du livre suivi de son prix entre parenthèse.

On utilisera également des couleurs de fond différentes pour les lignes paires et impaires (avec les styles CSS) et on affichera une petite image pour tous les livres dont le prix est inférieur à 30 #.

Le résultat final pourrait ressembler à cela (les informations proviennent de la section Livre Java : http://java.developpez.com/livres/) :



Obtenu avec le code HTML suivant :

Liste HTML

```
<ul>
        <li class='pair'>
                Jakarta Struts - par la pratique (37.05 &euro;)
        </li>
        <li class='impair'>
                Swing : la synthèse (37.91 &euro;)
        </li>
        <li class='pair'>
                Jakarta Struts - précis & concis (O'Reilly) (8.55 &euro;)
                <img src='hot.png' alt='Moins de 30 &euro;'/>
        </li>
        <li class='impair'>
                Java, conception et déploiement J2EE (23.75 &euro;)
                <img src='hot.png' alt='Moins de 30 &euro;'/>
        </li>
        <li class='pair'>
                Programmation Orienté Aspect pour Java / J2EE (42.75 &euro;)
        </li>
        <li class='impair'>
                Cahiers du Programmeur Java 1 (21.85 &euro;)
                <img src='hot.png' alt='Moins de 30 &euro;'/>
        </li>
        <li class='pair'>
                Java & XSLT (40.85 &euro;)</li>
        <li class='impair'>
                Java & XML 2nd Edition (47.5 &euro;)</li>
        <li class='pair'>
                Enterprise JavaBeans (45.6 &euro;)</li>
        <li class='impair'>
                Le Livre de Java premier langage (27.5 &euro;)
                <img src='hot.png' alt='Moins de 30 &euro;'/>
        </li>
        <li class='pair'>
                Java en action (55.1 &euro;)</li>
        <li class='impair'>
                Java in a Nutshell (51.3 &euro;)</li>
```

Liste HTML
```
            <li class='pair'>
                    Au coeur de Java 2 ); tome 1 (38.0 &euro;)</li>
            <li class='impair'>
                    Total Java (10.0 &euro;)
                    <img src='hot.png' alt='Moins de 30 &euro;'/>
            </li>
    </ul>
```

Il reste donc à générer ce code dynamiquement en utilisant la **List** "**books-list**" du **request**.

2.2.1 - Utilisation des scriptlets

Avec les **scriptlets**, la solution la plus simple est de récupérer la **List** et de la parcourir avec un **Iterator**. Ce qui nous donne :

Affichage d'une liste avec les scriptlets
```
    <%@ page import="packageNames.Book" %>
    <%@ page import="java.util.*" %>
    <ul>
    <% List bookList = (List) request.getAttribute ("books-list");
            if (bookList!=null) {
                    Iterator iterator = bookList.iterator();
                    int i = 0;
                    while ( iterator.hasNext() ) {
                            Book b = (Book) iterator.next();
                            out.print ("<li class='" + (i%2==0?"pair":"impair") + "'>");
                            out.print (b.getName() + " (" + b.getPrice() + " &euro;)");
                            if ( b.getPrice() < 30.0 )
                                    out.print (" <img src='hot.png' alt='Moins de 30 &euro;'/>");
                            out.println ("</li>");
                            i++;
                    }
            }
    %>
    </ul>
```

Le code fonctionne très bien, mais on perd tous les avantages des JSP puisqu'il n'y a quasiment que du code Java, et on conserve donc l'utilisation fastidieuse des **out.print()** (on pourrait bien sûr les remplacer par du code HTML, mais on se retrouve alors avec une multitude d'ouvertures/fermetures de scriptlets peu pratique...).

2.2.2 - Utilisation des EL

Les **EL** étant conçus pour fonctionner avec des tags JSP, on utilisera donc dans ce cas deux tags de la **JSTL** :

<c:forEach/>

Ce tag permet d'effectuer des itérations sur des collections. On l'utilisera avec les attributs suivants :

items : La collection d'objet (notre liste).

var : Nom de la variable qui contiendra les éléments successifs de la collection.

varStatus : Nom de la variable qui contiendra des informations concernant la collection et son itération (un objet qui contient entre autre la propriété **index** contenant le numéro d'index de l'élément courant).

<c:if/>

Ce tag permet d'effectuer une condition tels que le ferais le mot-clef **if**. On l'utilise l'attribut suivant :

test : La condition qui détermine si le corps du tag doit être exécuté ou pas...

Ce qui nous donne le code suivant :

Affichage d'une liste avec les EL et JSTL

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<ul>
<c:forEach items="${requestScope['books-list']}" var="book" varStatus="status">
        <li class="${status.index%2==0?'pair':'impair'}">${book.name} (${book.price} &euro;)
                <c:if test="${book.price < 30.0}">
                        <img src='hot.png' alt='Moins de 30 &euro;'/>
                </c:if>
        </li>
</c:forEach>
</ul>
```

On conserve le code HTML, et ce dernier est enrichi de nouvelles balises (les tags de la JSTL) et d'**EL** qui permettent de modifier dynamiquement la page selon les données reçues...

Conclusion

Les **EL** permettent de simplifier le code des pages JSP tout en améliorant la sécurité du code grâce à la gestion de certaine exception de base.

La notion d'**Expressions Languages** a été introduite afin de faciliter la conception de pages JSP, en particulier afin de pouvoir accéder et utiliser des données sans devoir maîtriser un langage aussi complexe que Java...

En effet, la logique de conception des pages JSP se rapproche de la logique de conception d'une page HTML ou d'un fichier XML. Ainsi, une formation de base peut permettre à un web designer de concevoir des pages JSP dynamiques en utilisant des tags/beans crées par un développeur Java...