Expression Language (EL)

Les expressions EL permettent via une syntaxe très épurée d'effectuer des tests basiques sur des expressions, et de manipuler simplement des objets et attributs dans une page, et cela sans nécessiter l'utilisation de code ni de script Java. La maintenance des pages JSP, en fournissant des notations simples et surtout standard, est ainsi grandement facilitée.

- ***Réalisation de test :***
  - Opérateurs arithmétiques, applicables à des nombres : +, -, *, /, % ;
  - Opérateurs logiques, applicables à des booléens : &&, ||, ! ;
  - Opérateurs relationnels, basés sur l'utilisation des méthodes `equals()` et `compareTo()` des objets comparés : == ou eq, != ou ne, < ou lt, > ou gt, <= ou le, >= ou ge.

Exemple : Code JSP :

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8" />
<title>Test des expressions EL</title>
</head>
<body>
<p>
<!-- Logiques sur des booléens -->
${ true && true } <br /> <!-- Affiche true -->
${ true && false } <br /> <!-- Affiche false -->
${ !true || false } <br /> <!-- Affiche false -->
<!-- Calculs arithmétiques -->
${ 10 / 4 } <br /> <!-- Affiche 2.5 -->
${ 10 mod 4 } <br /> <!-- Affiche le reste de la division
entière, soit 2 -->
${ 10 % 4 } <br /> <!-- Affiche le reste de la division
entière, soit 2 -->
${ 6 * 7 } <br /> <!-- Affiche 42 -->
${ 63 - 8 } <br /> <!-- Affiche 55 -->
${ 12 / -8 } <br /> <!-- Affiche -1.5 -->
${ 7 / 0 } <br /> <!-- Affiche Infinity -->
<!-- Compare les caractères 'a' et 'b'. Le caractère 'a'
étant bien situé avant le caractère 'b' dans l'alphabet ASCII,
cette EL affiche true. -->
${ 'a' < 'b' } <br />
<!-- Compare les chaînes 'hip' et 'hit'. Puisque 'p' < 't',
cette EL affiche false. -->
${ 'hip' gt 'hit' } <br />
<!-- Compare les caractères 'a' et 'b', puis les chaînes
'hip' et 'hit'. Puisque le premier test renvoie true et le second
false, le résultat est false. -->
${ 'a' < 'b' && 'hip' gt 'hit' } <br />
<!-- Compare le résultat d'un calcul à une valeur fixe.
Ici, 6 x 7 vaut 42 et non pas 48, le résultat est false. -->
${ 6 * 7 == 48 } <br />
</p>
</body>
</html>
```

➢ ***Deux autres types de test sont fréquemment utilisés au sein des expressions EL :***

- Les conditions ternaires, de la forme : `test ? si oui : sinon ;`
- Les vérifications si vide ou **`null`**, grâce à l'opérateur **`empty`**.
  Exemple : Code JSP :

```
<!-- Vérifications si vide ou null -->
${ empty 'test' } <!-- La chaîne testée n'est pas vide, le résultat
est false -->
${ empty '' } <!-- La chaîne testée est vide, le résultat est true
-->
${ !empty '' } <!-- La chaîne testée est vide, le résultat est
false -->
<!-- Conditions ternaires -->
${ true ? 'vrai' : 'faux' } <!-- Le booléen testé vaut true, vrai
est affiché -->
${ 'a' > 'b' ? 'oui' : 'non' } <!-- Le résultat de la comparaison
vaut false, non est affiché -->
${ empty 'test' ? 'vide' : 'non vide' } <!-- La chaîne testée
n'est pas vide, non vide est affiché -->
```

La valeur retournée par une expression EL positionnée dans un texte ou un contenu statique sera insérée à l'endroit même où est située l'expression :

Exemple : Code JSP :

```
<!-- La ligne suivante : -->
<p>12 est inférieur à 8 : ${ 12 lt 8 }.</p>
<!-- Sera rendue ainsi après interprétation de l'expression, 12 n'étant

 pas inférieur à 8 : -->
<p>12 est inférieur à 8 : false.</p>
```

➢ ***La manipulation d'objets :***

La vraie puissance des expressions EL, leur véritable intérêt, c'est le fait qu'elles permettent de manipuler des objets et de leur appliquer tous ces tests.

- ***DES BEANS :***

Exemple : En remplaçant le code JSP concernant les tests standards précédents par le code suivant :

```
<!DOCTYPE html>
   <html>
   <head>
    <meta charset="utf-8" />
    <title>Test des expressions EL</title>
   </head>
   <body>
    <p>
     <!-- Initialisation d'un bean de type Coyote

       avec une action standard,  pour  l'exemple : -->
   <jsp:useBean id="coyote" class="com.sdzee.beans.Coyote" />
   <!-- Initialisation de sa propriété 'prénom' : -->
   <jsp:setProperty name="coyote" property="prenom" value="FIDY"/>
   <!-- Et affichage de sa valeur : -->
  <jsp:getProperty name="coyote" property="prenom" />
   </p>
  </body>
  </html>
```

Grâce aux deux premières actions (lignes 10 et 12), la base de notre exemple est posée : nous créons un bean de type **Coyote** dans notre page JSP, et initialisons sa propriété **prenom** avec la valeur "FIDY".

Maintenant, remplacez cette ligne 14 par l'expression EL suivante :

```
${ coyote.prenom }
```

Actualisez la page de tests dans votre navigateur, et vous observerez que le contenu n'a pas changé, la valeur de la propriété prénom est toujours correctement affichée. C'est tout ce qu'il est nécessaire d'écrire avec la technologie EL. Cette expression retourne le contenu de la propriété **prenom** du bean nommé **coyote**. Remarquez bien la syntaxe et la convention employées :

- **coyote** est le nom du bean, que nous avions ici défini dans l'attribut **id** de l'action **`<jsp:useBean>`** ;
- **prenom** est un champ privé du bean (une propriété) accessible par sa méthode publique `getPrenom()` ; l'opérateur *point* permet de séparer le bean visé de sa propriété.

Ainsi de manière générale, il suffit d'écrire `${ bean.propriete }` pour accéder à une propriété d'un bean.

Pour information, voici ce à quoi ressemble le code Java qui est mis en œuvre dans les coulisses lors de l'interprétation de l'expression `${ coyote.prenom }` :

Code JAVA :

```
Coyote bean = (Coyote) pageContext.findAttribute( "coyote" );
if ( bean != null ) {
String prenom = bean.getPrenom();
if ( prenom != null ) {
out.print( prenom );
 }
}
```

Peu importe que nous la comparions avec une scriptlet Java ou avec une action standard, l'expression EL **simplifie l'écriture de manière frappante.** Ci-dessous, on se propose d'étudier quelques exemples des utilisations et erreurs de syntaxe les plus courantes :

Code JSP :

```
<!-- Syntaxe conseillée pour récupérer la propriété 'prenom' du
bean 'coyote'. -->
${ coyote.prenom }
<!-- Syntaxe correcte, car il est possible d'expliciter la méthode
d'accès à la propriété. Préférez toutefois la notation précédente.
-->
${ coyote.getPrenom() }
<!-- Syntaxe erronée : la première lettre de la propriété doit être
une minuscule. -->
${ coyote.Prenom }
```

Voilà pour la syntaxe à employer pour accéder à des objets. Voyons maintenant que la vraie puissance de la technologie EL réside non seulement dans le fait qu'elle permet de manipuler des beans, mais également dans le fait qu'elle permet de les faire intervenir au sein de tests. Voici quelques exemples d'utilisations, toujours basés sur la propriété **prenom** du bean **coyote** :

Code JSP :

```
<!-- Comparaison d'égalité entre la propriété prenom et la chaîne
"Jean-Paul" -->
${ coyote.prenom == "Jean-Paul" }
<!-- Vérification si la propriété prenom est vide ou nulle -->
${ empty coyote.prenom }
<!-- Condition ternaire qui affiche la propriété prénom si elle
n'est ni vide ni nulle, et la chaîne "Veuillez préciser un prénom"
sinon -->
${ !empty coyote.prenom ? coyote.prenom : "Veuillez préciser un
prénom" }
```

En outre, il faut savoir également que **les expressions EL sont protégées contre un éventuel retour `null`** :

Code JSP :

```
<!-- La scriptlet suivante affiche "null" si la propriété "prenom"
n'a pas été initialisée,
et provoque une erreur à la compilation si l'objet "coyote" n'a pas
été initialisé : -->
<%= coyote.getPrenom() %>
<!-- L'action suivante affiche "null" si la propriété "prenom" n'a
pas été initialisée,
et provoque une erreur à l'exécution si l'objet "coyote" n'a pas
été initialisé : -->
<jsp:getProperty name="coyote" property="prenom" />
<!-- L'expression EL suivante n'affiche rien si la propriété
"prenom" n'a pas été initialisée,
et n'affiche rien si l'objet "coyote" n'a pas été initialisé : -->
${ coyote.prenom }
```

- ***DES COLLECTIONS***

Les beans ne sont pas les seuls objets manipulables dans une expression EL, il est également possible d'accéder aux collections au sens large. Cela inclut donc tous les objets de type `java.util.List`, `java.util.Set`, `java.util.Map`, etc.
Pour commencer, on va reprendre notre page **JSP**, et remplacer son code par cet exemple illustrant l'accès aux éléments d'une liste :

Code JSP :

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8" />
<title>Test des expressions EL</title>
</head>
<body>
<p>
<%
/* Création d'une liste de légumes et insertion de quatre
éléments */
java.util.List<String> legumes = new
java.util.ArrayList<String>();
legumes.add( "poireau" );
legumes.add( "haricot" );
legumes.add( "carotte");
legumes.add( "pomme de terre" );
request.setAttribute( "legumes" , legumes );
%>
<!-- Les quatre syntaxes suivantes retournent le deuxième
élément de la liste de légumes -->
${ legumes.get(1) }<br />
${ legumes[1] }<br />
${ legumes['1'] }<br />
${ legumes["1"] }<br />
</p>
</body>
</html>
```

On découvre aux lignes 20 à 23 quatre nouvelles notations :

- l'appel à une méthode de l'objet **legumes**. En l'occurrence notre objet est une liste, et nous appelons sa méthode `get()` en lui passant l'indice de l'élément voulu ;
- l'utilisation de crochets à la place de l'opérateur *point*, contenant directement l'indice de l'élément voulu ;
- l'utilisation de crochets à la place de l'opérateur *point*, contenant l'indice de l'élément entouré d'apostrophes ;
- l'utilisation de crochets à la place de l'opérateur *point*, contenant l'indice de l'élément entouré de guillemets

Remplacez ensuite le code de l'exemple par le suivant, illustrant l'accès aux éléments d'un tableau :

Code JSP :

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <title>Test des expressions EL</title>
</head>
<body>
  <p>
  <%
/* Création d'un tableau */
    String[] animaux = {"chien", "chat", "souris", "cheval"};
    request.setAttribute("animaux" , animaux);
  %>
  <!-- Les trois syntaxes suivantes retournent le troisième
  élément du tableau -->
  ${ animaux[2] }<br />
  ${ animaux['2'] }<br />
  ${ animaux["2"] }<br />
  </p>
</body>
</html>
```

Sachez par ailleurs qu'il est impossible d'utiliser directement l'opérateur *point* pour accéder à un élément d'une liste ou d'un tableau. Les syntaxes `${entiers.1}` ou `${animaux.2}` enverront une exception à l'exécution.

Enfin, remplacez le code de l'exemple par le suivant, illustrant l'accès aux éléments d'une `Map` :

Code JSP :

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <title>Test des expressions EL</title>
</head>
<body>
  <p>
  <%
/* Création d'une Map */
java.util.Map<String,Integer> desserts = new
java.util.HashMap<String, Integer>();
desserts.put("cookies", 8);
desserts.put("glaces", 3);
desserts.put("muffins", 6);
desserts.put("tartes aux pommes", 2);
request.setAttribute("desserts" , desserts);
%>
<!-- Les quatre syntaxes suivantes retournent la valeur
associée à la clé "cookies" de la Map de desserts -->
${ desserts.cookies }<br />
${ desserts.get("cookies") }<br />
${ desserts['cookies'] }<br />
${ desserts["cookies"] }<br />
<%
/* Création d'une chaîne nommée "element" et contenant le
mot "cookies" */
String element = "cookies";
request.setAttribute("element",element);
%>
<!-- Il est également possible d'utiliser un objet au lieu
d'initialiser la clé souhaitée directement dans l'expression -->
```

```
${ desserts[element] }<br />
</p>
</body>
</html>
```

Rendez-vous une nouvelle fois sur la page de tests depuis votre navigateur, et remarquez deuxchoses importantes :

- La notation avec l'opérateur *point* fonctionne, à la ligne 21, de la même manière que lors de l'accès à une propriété d'un bean ;
- Une expression peut en cacher une autre. En l'occurrence à la ligne 32, lorsque le conteneur va analyser l'expression EL il va d'abord récupérer l'attribut de requête **element**, puis utiliser son contenu en guise de clé afin d'accéder à la valeur
  associée dans la `Map` de desserts.

Par ailleurs, faites attention à la syntaxe `${desserts[element]}` employée dans ce dernier cas :

- il est impératif de ne pas entourer le nom de l'objet d'apostrophes ou de guillemets au sein des crochets. En effet, écrire`${desserts["element"]}` reviendrait à essayer d'accéder à la valeur associée à la clé nommée "element", ce quin'est pas le comportement souhaité ici ;
- il ne faut pas entourer l'expression contenue dans l'expression englobante avec des accolades. Autrement dit, il ne faut pas écrire `${desserts[${element}]}`, cette syntaxe n'est pas valide. Une seule paire d'accolades suffit

➢ ***Désactiver l'évaluation des expressions EL***

La directive suivante désactive l'évaluation des EL dans une page JSP :

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

Les seules valeurs acceptées par l'attribut **isELIgnored** sont **`true`** et **`false`** :

- s'il est initialisé à **`true`**, alors les expressions EL seront ignorées et apparaîtront en tant que simples chaînes de caractères ;
- s'il est initialisé à **`false`**, alors elles seront interprétées par le conteneur.

On peut d'ailleurs faire le test. Éditez le fichier JSP et insérez-y en première ligne la directive.
Enregistrez la modification et actualisez alors l'affichage de la page dans votre navigateur. Vous observerez que vos expressions ne sont plus évaluées, elles sont cette fois simplement affichées telles quelles.

- ***AVEC LE FICHIER web.xml***
  V ous pouvez désactiver l'évaluation des expressions EL sur tout un ensemble de pages JSP dans une application grâce à l'option **`<el-ignored>`** du fichier web.xml.

Code XML :

```
<jsp-config>
   <jsp-property-group>
   <url-pattern>*.jsp</url-pattern>
   <el-ignored>true</el-ignored>
   </jsp-property-group>
</jsp-config>
```

Si **`true`** est précisé alors les EL seront ignorées, et si **`false`** est précisé elles seront interprétées.

➢ ***Les objets implicites :***

Il nous reste un concept important à aborder avant de passer à la pratique : les objets implicites. Il en existe deux types :

- Ceux qui sont mis à disposition via la technologie JSP ;
- Ceux qui sont mis à disposition via la technologie EL.

➢ ***Les objets de la technologie JSP :***

Pour illustrer ce nouveau concept, revenons sur la première JSP que nous avions écrite dans le chapitre sur la transmission des données :

Code : JSP - /WEB-INF/test.jsp

```
<%@ page pageEncoding="UTF-8" %>
<!DOCTYPE html>
<html>
 <head>
   <meta charset="utf-8" />
   <title>Test</title>
</head>
<body>
   <p>Ceci est une page générée depuis une JSP.</p>
   <p>
   <%
    String attribut = (String) request.getAttribute("test");
    out.println( attribut );
   %>
   </p>
  </body>
  </html>
```

On remarque qu'à la ligne 13, nous avions directement utilisé l'objet **out** sans jamais l'avoir instancié auparavant. De même, à la ligne 12 nous accédions directement à la méthode `request.getAttribute()` sans jamais avoir instancié d'objet nommé **request**…

Nous devons nous intéresser une nouvelle fois au code de la servlet auto-générée par Tomcat. Retournons dans le répertoire **work** du serveur, qui est subtilisé par Eclipse, et analysons à nouveau le code du fichier **Test.java** :

Code : Java - Extrait de la servlet auto-générée par Tomcat

```
...
public void _jspService(final
  javax.servlet.http.HttpServletRequest request, final
  javax.servlet.http.HttpServletResponse response)
throws java.io.IOException, javax.servlet.ServletException {
final javax.servlet.jsp.PageContext pageContext;
  javax.servlet.http.HttpSession session = null;
final javax.servlet.ServletContext application;
final javax.servlet.ServletConfig config;
javax.servlet.jsp.JspWriter out = null;
final java.lang.Object page = this;
javax.servlet.jsp.JspWriter _jspx_out = null;
javax.servlet.jsp.PageContext _jspx_page_context = null;
try {
response.setContentType("text/html");
pageContext = _jspxFactory.getPageContext(this, request,
response,
null, true, 8192, true);
_jspx_page_context = pageContext;
application = pageContext.getServletContext();
config = pageContext.getServletConfig();
session = pageContext.getSession();
out = pageContext.getOut();
_jspx_out = out;
out.write("<!DOCTYPE html>\r\n");
out.write("<html>\r\n");
out.write(" <head>\r\n");
out.write(" <meta charset=\"utf-8\" />\r\n");
out.write(" <title>Test</title>\r\n");
out.write(" </head>\r\n");
out.write(" <body>\r\n");
out.write(" <p>Ceci est une page générée depuis une
JSP.</p>\r\n");
out.write(" <p>\r\n");
out.write(" ");
String attribut = (String) request.getAttribute("test");
```

```
out.println( attribut );
out.write("\r\n");
out.write(" </p>\r\n");
out.write(" </body>\r\n");
out.write("</html>");
}
...
```

Analysons ce qui se passe dans le cas de l'objet **out** :

- à la ligne 10, un objet nommé **out** et de type `JspWriter` est créé ;
- à la ligne 24, il est initialisé avec l'objet *writer* récupéré depuis la réponse ;
- à la ligne 39, c'est tout simplement notre ligne de code Java, basée sur l'objet **out**, qui est recopiée telle quelle de la JSP vers la servlet auto-générée.

Le conteneur s'en charge pour nous lorsqu'il traduit votre page en servlet :

Identifiant	Type de l'objet	Description
pageContext	`PageContext`	Il fournit des informations utiles relatives au contexte d'exécution. Entre autres, il permet d'accéder aux attributs présents dans les différentes portées de l'application. Il contient également une référence vers tous les objets implicites suivants.
application	`ServletContext`	Il permet depuis une page JSP d'obtenir ou de modifier des informations relatives à l'application dans laquelle elle est exécutée.
session	`HttpSession`	Il représente une session associée à un client. Il est utilisé pour lire ou placer des objets dans la session de l'utilisateur courant.
request	`HttpServletRequest`	Il représente la requête faite par le client. Il est généralement utilisé pour accéder aux paramètres et aux attributs de la requête, ainsi qu'à ses en-têtes.
response	`HttpServletResponse`	Il représente la réponse qui va être envoyée au client. Il est généralement utilisé pour définir le Content-Type de la réponse, lui ajouter des en-têtes ou encore pour rediriger le client.
exception	`Throwable`	Il est uniquement disponible dans les pages d'erreur JSP. Il représente l'exception qui a conduit à la page d'erreur en question.
out	`JspWriter`	Il représente le contenu de la réponse qui va être envoyée au client. Il est utilisé pour écrire dans le corps de la réponse.
config	`ServletConfig`	Il permet depuis une page JSP d'obtenir les éventuels paramètres d'initialisation disponibles.
page	objet `this`	Il est l'équivalent de la référence `this` et représente la page JSP courante. Il est déconseillé de l'utiliser, pour des raisons de dégradation des performances notamment.

- ***Les objets de la technologie EL***

  Nous allons grâce à elle pouvoir profiter des objets implicites sans écrire de code Java : il s'agit d'objets gérés automatiquement par le conteneur lors de l'évaluation des expressions EL, et auxquels nous pouvons directement accéder depuis nos expressions sans les déclarer auparavant :

Catégorie	Identifiant	Description
JSP	**pageContext**	Objet contenant des informations sur l'environnement du serveur.
Portées	**pageScope**	Une `Map` qui associe les noms et valeurs des attributs ayant pour portée la page.
	requestScope	Une `Map` qui associe les noms et valeurs des attributs ayant pour portée la requête.
	sessionScope	Une `Map` qui associe les noms et valeurs des attributs ayant pour portée la session.
	applicationScope	Une `Map` qui associe les noms et valeurs des attributs ayant pour portée l'application.
Paramètres de requête	**param**	Une `Map` qui associe les noms et valeurs des paramètres de la requête.
	paramValues	Une `Map` qui associe les noms et multiples valeurs ** des paramètres de la requête sous forme de tableaux de String.
En-têtes de requête	**header**	Une `Map` qui associe les noms et valeurs des paramètres des en-têtes HTTP.
	headerValues	Une `Map` qui associe les noms et multiples valeurs ** des paramètres des en-têtes HTTP sous forme de tableaux de String.
Cookies	**cookie**	Une `Map` qui associe les noms et instances des cookies.
Paramètres d'initialisation	**initParam**	Une `Map` qui associe les données contenues dans les champs `<param-name>` et `<param-value>` de la section `<init-param>` du fichier web.xml.

La première chose à remarquer dans ce dernier tableau, c'est que le seul objet implicite en commun entre les JSP et les expressions EL est le **pageContext**.
La seconde, c'est la différence flagrante avec les objets implicites JSP : tous les autres objets implicites de la technologie EL sont des `Map`
Ça peut paraître compliqué, mais en réalité c'est très simple. C'est un outil incontournable en Java, et nous venons d'en manipuler une lorsque nous avons découvert les expressions EL, sachez qu'une `Map` est un objet qui peut se représenter comme un tableau à deux colonnes :

- la première colonne contient ce que l'on nomme les **clés**, qui doivent obligatoirement être uniques ;
- la seconde contient les valeurs, qui peuvent quant à elles être associées à plusieurs clés.

Chaque ligne du tableau ne peut contenir qu'une clé et une valeur. Voici un exemple d'une `Map<String, String>` représentant une liste d'aliments et leurs types :

Aliments (Clés)	Types (Valeurs)
pomme	fruit
carotte	légume
boeuf	viande
aubergine	légume
…	…

Vous voyez bien ici qu'un même type peut être associé à différents aliments, mais qu'un même aliment ne peut exister qu'une seule fois dans la liste. Eh bien c'est ça le principe d'une `Map` : c'est un ensemble d'éléments uniques auxquels on peut associer n'importe quelle valeur.

Créons une page **test_map.jsp**, dans laquelle nous allons implémenter rapidement cette `Map` d'aliments :

Code JSP : test_map.jsp

```
<%@ page import="java.util.Map, java.util.HashMap" %>
   <!DOCTYPE html>
   <html>
   <head>
     <meta charset="utf-8" />
     <title>Test des Maps et EL</title>
   </head>
   <body>
     <p>
     <%
     Map<String, String> aliments = new HashMap<String,
     String>();
     aliments.put( "pomme","fruit" );
     aliments.put( "carotte","légume" );
     aliments.put( "boeuf","viande" );
     aliments.put( "aubergine","légume" );
     request.setAttribute( "aliments", aliments );
     %>
      ${ aliments.pomme } <br /> <!-- affiche fruit -->
      ${ aliments.carotte } <br /> <!-- affiche légume -->
      ${ aliments.boeuf } <br /> <!-- affiche viande -->
      ${ aliments.aubergine } <br /><!-- affiche légume -->
     </p>
   </body>
   </html>
```

Le rapport avec les objets impilites EL, c'est que tous ces objets sont des `Map`, et que par conséquent nous sommes capables d'y accéder depuis des expressions EL, de la même manière que nous venons de parcourir notre `Map` d'aliments. Pour illustrer le principe, nous allons laisser tomber nos fruits et légumes et créer une page nommée **test_obj_impl.jsp**, encore et toujours à la racine de notre session, application, et y placer le code suivant :

Code : JSP - /test_obj_impl.jsp

```
<!DOCTYPE html>
<html>
  <head>
   <meta charset="utf-8" />
   <title>Test des objets implicites EL</title>
  </head>
<body>
 <p>
<%
 String paramLangue = request.getParameter("langue");
 out.println( "Langue : " + paramLangue );
%>
 <br />
<%
  String paramArticle = request.getParameter("article");
  out.println( "Article : " + paramArticle );
%>
 </p>
</body>
 </html>
```

On reconnait aux lignes 10 et 15 la méthode `request.getParameter()` permettant de récupérer les paramètres transmis au serveur par le client à travers l'URL. Ainsi, il suffit par exemple de se rendre sur http://localhost:8080/test/test_obj_impl.jsp?langue=fr&article=782 pour que votre navigateur vous affiche ceci (voir la figure suivante) :

On peut chercher maintenant, dans le tableau fourni précédemment, l'objet implicite EL dédié à l'accès aux paramètres de requête… Il s'agit de la Map nommée param. La technologie EL va ainsi vous mettre à disposition un objet dont le contenu peut, dans le cas de notre exemple, être représenté sous cette forme :

Nom du paramètre (Clé)	Valeur du paramètre (Valeur)
langue	fr
article	782

De même que l'exemple avec les fruits et légumes, on fait également pour accéder à nos paramètres de requêtes depuis des expressions EL. Éditons notre fichier **test_obj_impl.jsp** et remplaçons le code précédent par le suivant :

Code : JSP - /test_obj_impl.jsp

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <title>Test des objets implicites EL</title>
</head>
<body>
<p>
  Langue : ${ param.langue }
  <br />
  Article : ${ param.article }
</p>
</body>
</html>
```

Amélioration de vue

Pour rappel, voici où nous en étions après l'introduction d'un bean dans notre exemple :
Code : JSP - /WEB-INF/test.jsp

```
<%@ page pageEncoding="UTF-8" %>
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>Test</title>
  </head>
 <body>
  <p>Ceci est une page générée depuis une JSP.</p>
   <p>
   <%
   String attribut = (String) request.getAttribute("test");
   out.println( attribut );
   String parametre = request.getParameter( "auteur" );
   out.println( parametre );
   %>
   </p>
  <p>
  Récupération du bean :
  <%
   com.sdzee.beans.Coyote notreBean = (com.sdzee.beans.Coyote)
   request.getAttribute("coyote");
   out.println( notreBean.getPrenom() );
   out.println( notreBean.getNom() );
```

```
  %>
 </p>
 </body>
 </html>
```

Avec tout ce que nous avons appris, nous sommes maintenant capables de modifier cette page JSP pour qu'elle ne contienne plus de langage Java !
Pour bien couvrir l'ensemble des méthodes existantes, divisons le travail en deux étapes : avec des scripts et balises JSP pour commencer, puis avec des EL

- ***Avec des scripts et balises JSP***

Code : JSP - /WEB-INF/test.jsp

```
<%@ page pageEncoding="UTF-8" %>
<!DOCTYPE html>
 <html>
   <head>
     <meta charset="utf-8" />
     <title>Test</title>
   </head>
<body>
<p>Ceci est une page générée depuis une JSP.</p>
<p>
<% String attribut = (String)
request.getAttribute("test"); %>
<%= attribut %>
<% String parametre = request.getParameter( "auteur" );
%>
<%= parametre %>
</p>
<p>
Récupération du bean :
<jsp:useBean id="coyote" class="com.sdzee.beans.Coyote"
scope="request" />
<jsp:getProperty name="coyote" property="prenom" />
<jsp:getProperty name="coyote" property="nom" />
</p>
</body>
</html>
```

On peut remarquer :

- L'affichage de l'attribut et du paramètre via la balise d'expression **<%= ... %>** ;
- La récupération du bean depuis la requête via la balise **<jsp:useBean>** ;
- L'affichage du contenu des propriétés via les balises **<jsp:getProperty>**.

- ***Avec des EL***

Code : JSP - /WEB-INF/test.jsp

```
<%@ page pageEncoding="UTF-8" %>
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8" />
<title>Test</title>
</head>
<body>
<p>Ceci est une page générée depuis une JSP.</p>
<p>
${test}
${param.auteur}
</p>
<p>
Récupération du bean :
${coyote.prenom}
```

```
${coyote.nom}
</p>
</body>
</html>
```

Enoncé du TP :

L'objectif, c'est de vous familiariser avec le développement web sous Eclipse. Vous allez devoir mettre en place un projet en partant de zéro dans votre environnement, et y créer vos différents fichiers. Le second objectif est que vous soyez à l'aise avec l'utilisation de servlets, de pages JSP et de beans, et de manière générale avec le principe général d'une application Java EE.

➢ *Fonctionnalités :*

Création d'un client

À travers notre petite application, l'utilisateur doit pouvoir créer un client en saisissant des données depuis un formulaire, et visualiser la fiche client en résultant. Puisque vous n'avez pas encore découvert les formulaires, je vais vous fournir une page qui vous servira de base. V otre travail sera de coder :

- un bean, représentant un client ;
- une servlet, chargée de récupérer les données envoyées par le formulaire, de les enregistrer dans le bean et de lestransmettre à une JSP ;
- une JSP, chargée d'afficher la fiche du client créé, c'est-à-dire les données transmises par la servlet.

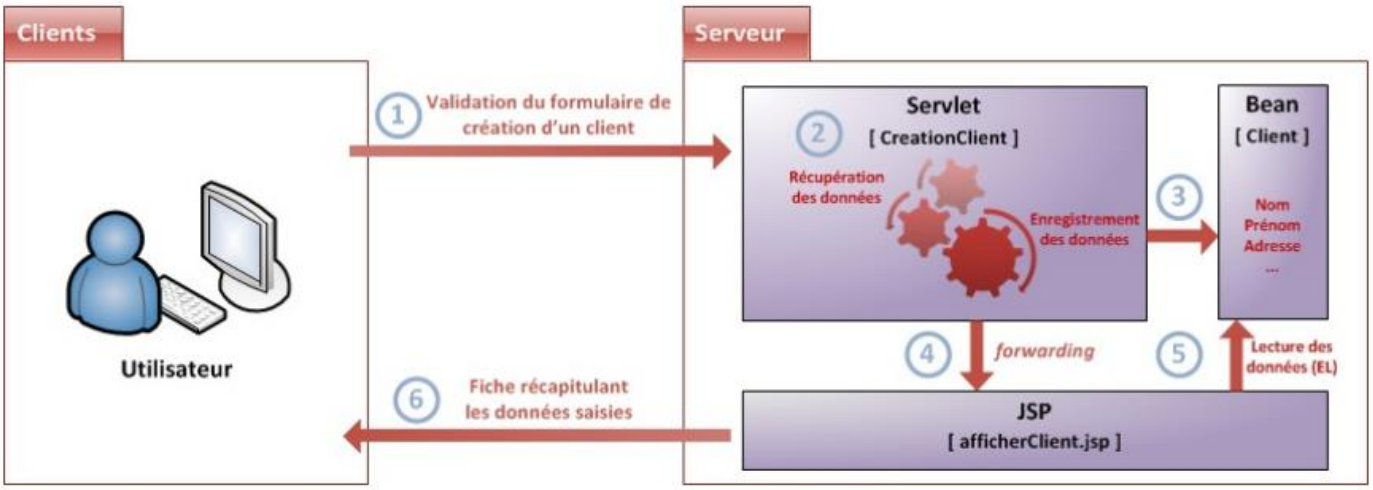
Création d'une commande

L'utilisateur doit également pouvoir créer une commande, en saisissant des données depuis un formulaire, et visualiser la fiche en résultant. De même, puisque vous n'avez pas encore découvert les formulaires, je vais vous fournir une page qui vous servira de base. V otre travail sera de coder :

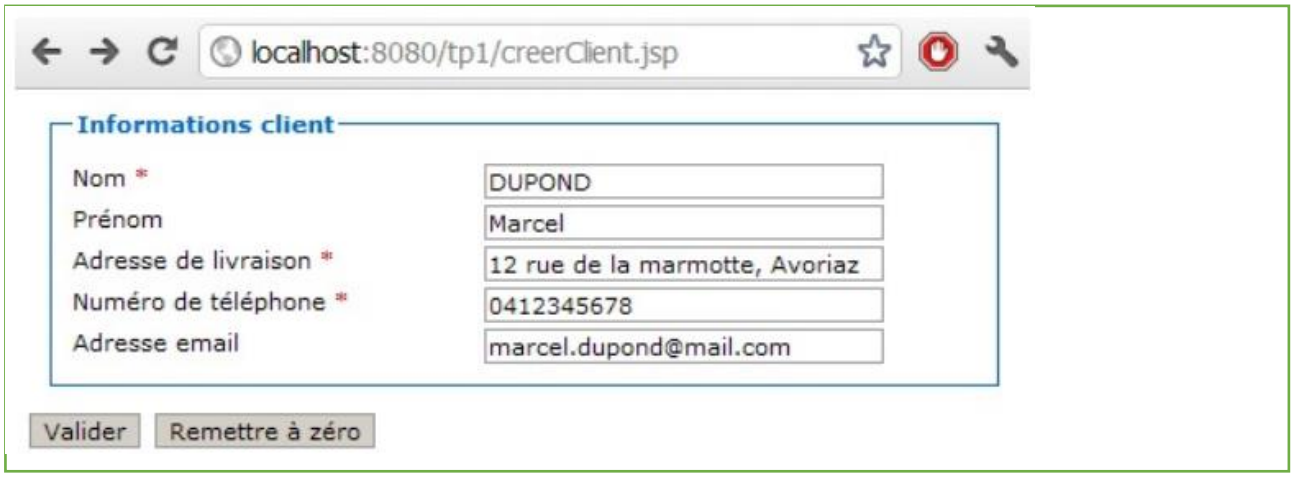
- un bean, représentant une commande ;
- une servlet, chargée de récupérer les données envoyées par le formulaire, de les enregistrer dans le bean et de lestransmettre à une JSP ;
- une JSP, chargée d'afficher la fiche de la commande créée, c'est-à-dire les données transmises par la servlet.

➢ *Illustration du comportement attendu*

À la figure suivante, voici sous forme d'un schéma ce que vous devez réaliser dans le cas de la création d'un client.

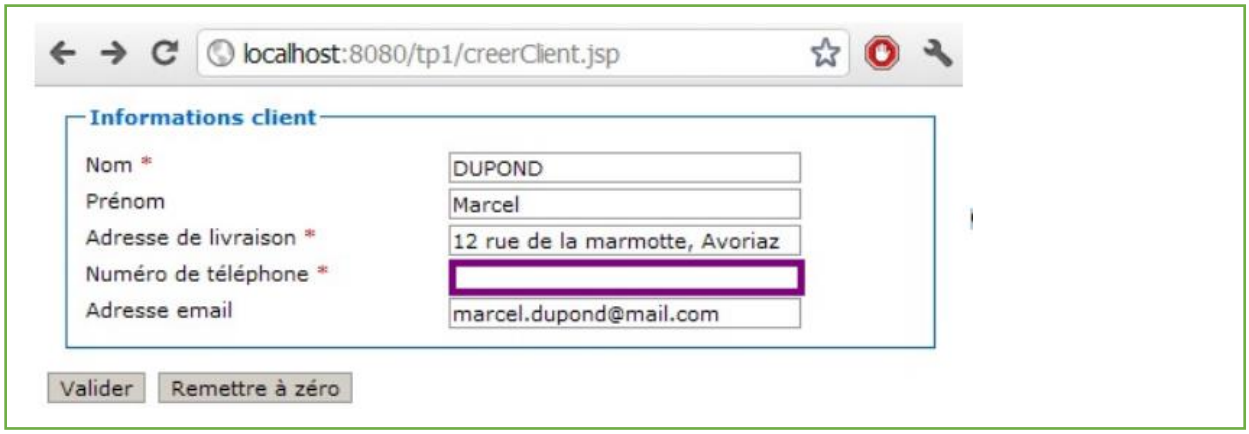


➢ ***Résultat :*** Saisie de données valides dans le formulaire



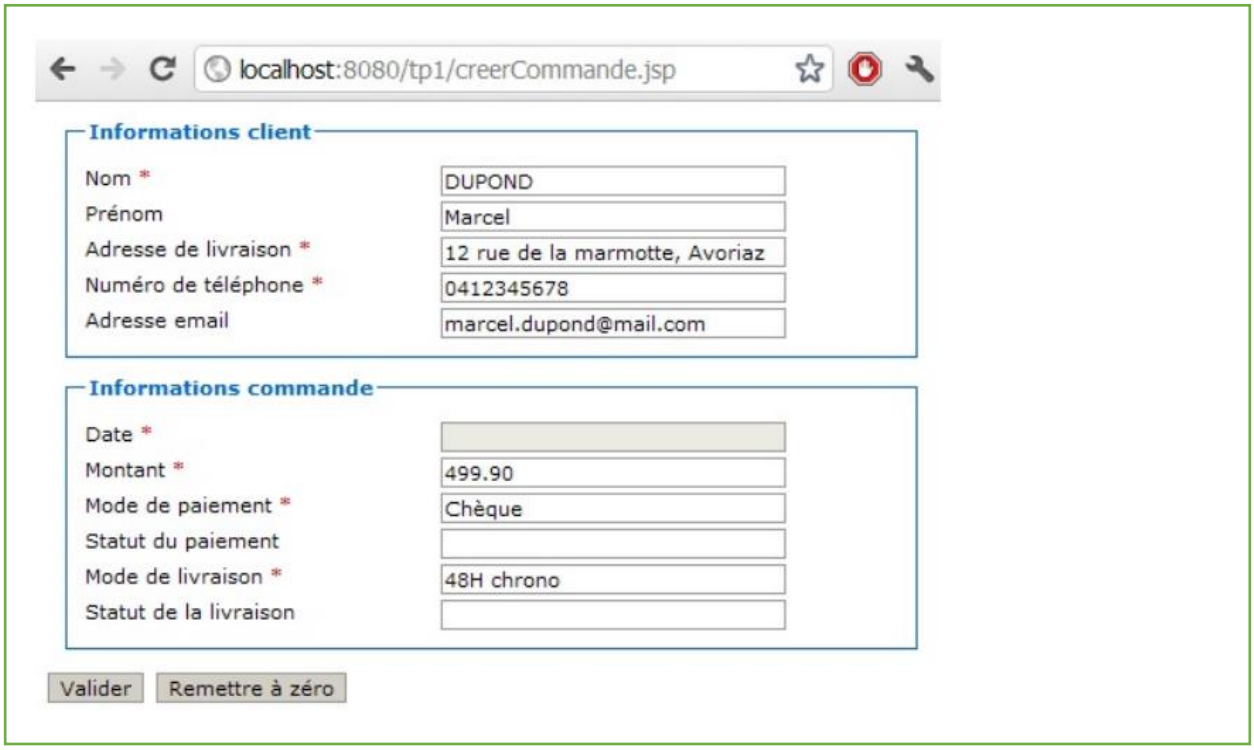
- ***Résultat :*** Affichage du message de succès et des données :

- ***Résultat :*** **Avec erreur :** Oubli d'un champ obligatoire dans le formulaire :

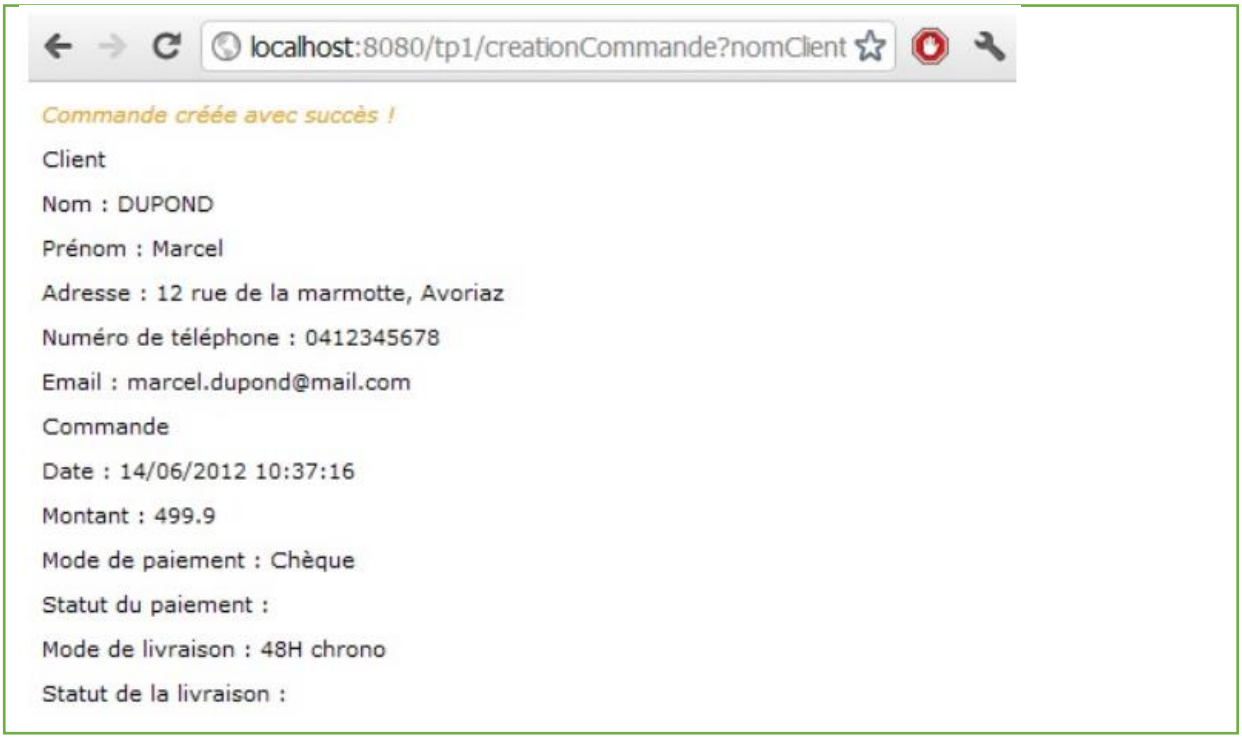


- ***Résultat :*** **Avec erreur :** Affichage du message d'erreur et des données :

- ***Création d'une commande (Succès) :*** Saisie de données valides dans le formulaire



- ***Résultat (Succès) :*** Affichage du message de succès et des données :



- ***Résultat (avec erreurs) :*** Oubli de champs obligatoires et saisie d'un montant erroné dans le formulaire

- ***Résultat (avec erreurs) :*** Affichage du message d'erreur et des données :