

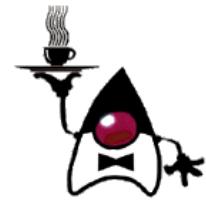
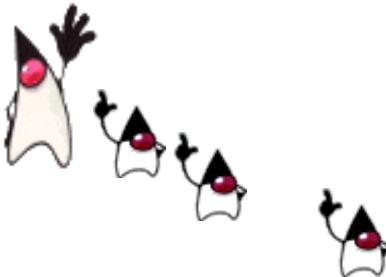
Java (III)
Programmation des applications Web

Claude VILVENS

claude.vilvens@hepl.be

<http://haute-ecole.provincedeliege.be/>

Sommaire



Introduction

XVI. Les servlets

1. L'extension des serveurs	
1.1 Le besoin et les solutions non Java	2
1.2 Les servlets	3
1.3 Les moteurs de servlets	4
2. Les packages pour servlets	4
3. Le cycle de vie d'une servlet	
3.1 L'initialisation	5
3.2 Le traitement des requêtes des clients	6
3.3 La terminaison d'une servlet	6
4. Le code d'une servlet élémentaire	6
5. Le moteur à servlets Tomcat	
5.1 Un moteur à servlets doublé d'un serveur HTTP	8
5.2 Tomcat en tant que serveur Web autonome (Windows)	8
5.3 L'architecture de Tomcat	14
5.4 Tomcat en tant que serveur Web autonome (Linux)	17
5.5 Tomcat dans NetBeans	18
5.6 Les applications Web et les Web application ARchive	19
5.7 Le descripteur de déploiement web.wml	21
6. Le développement d'une servlet avec NetBeans et Tomcat	
6.1 Mise en place du contexte Web	24
6.2 La création d'une servlet	25
6.3 L'exécution de la servlet	30
7. Le déploiement d'une servlet avec Tomcat standalone	
7.1 Le déploiement d'une servlet sous forme de fichier class	32
7.2 Le déploiement d'une servlet sous forme de WAR avec le contexte par défaut	34
7.3 Le déploiement d'une servlet sous forme de WAR avec un contexte spécifique	36
7.4 Le déploiement du module d'administration de Tomcat 5	37
7.5 Le déploiement d'une servlet sous forme de WAR selon le couple Tomcat-NetBeans	38
7.6 Le déploiement avec Tomcat Manager	39
7.7 Une visualisation du trafic réseau	41
8. Un seul objet servlet pour plusieurs requêtes	42
9. Une servlet d'informations	46

10. Une servlet pour une requête paramétrée (méthode GET)	
10.1 La récupération des données d'un formulaire	47
10.2 Un test en local avec NetBeans	49
10.3 Un test à distance sur une machine Linux	53
11. Une servlet pour une requête paramétrée (méthode POST)	53
12. Une servlet avec JDBC	54
13. Les paramètres d'initialisation d'une servlet	
13.1 L'initialisation d'une servlet	57
13.2 Les paramètres d'initialisation pour Tomcat	57
13.3 Une servlet avec paramètres d'initialisation	59
14. La communication applet-servlet	
14.1 La communication par sockets	63
14.2 La communication par http	64
15. Un formulaire dans une applet : la méthode GET	
15.1 L'utilisation d'une applet en local	64
15.2 Le déploiement de l'applet dans l'application Web	67
15.3 L'exécution de l'application Web	70
15.4 Le monitoring http avec NetBeans	71
15.4 L'utilisation d'une applet avec un serveur distant	74
15.5 Un environnement différent : Linux-Apache-Tomcat-MySQL	76
16. Une véritable communication applet-servlet : la méthode POST	
16.1 Un tunnel http	80
16.2 L'applet du tunnel http	81
16.3 La servlet du tunnel http	84
16.4 Le test du tunnel HTTP	87
16.5 Du point de vue du trafic réseau	90
16.6 Le modèle multithread	91
17. La communication applet-servlet par objets sérialisés	
17.1 Un package global	93
17.2 L'applet de la communication sérialisée	94
17.3 La servlet de la communication sérialisée	96
18. Les cookies	
18.1 Le principe	98
18.2 La création des cookies en Java	98
18.3 Ecrire et lire des cookies	99
18.4 Un cookie pour la dernière visite	99
18.5 Un cookie pour l'identification d'une session	103
19. Le suivi de session	
19.1 L'interface HttpSession	106
19.2 Les classiques : l'id de session et le compteur d'accès	107
19.3 Le cycle de vie d'une session	112
20. L'authentification d'un client	
20.1 Les acteurs	113
20.2 L'applet de connexion	113
20.3 La servlet d'accès	115
20.4 La servlet de contrôle	117
21. Le ServletContext	120

XVII. Les Java Server Pages

1. La technique SSI	124
2. Les JSP	125
3. Un exemple introductif	125
4. Le cycle de vie d'un JSP	128
5. Servlets ou JSP ?	131
6. Les objets implicites	131
7. Un classique : le traitement d'un formulaire	
7.1 La création du JSP avec NetBeans	132
7.2 La création de la page formulaire	136
7.3 L'exécution de l'application en local avec Tomcat intégré à NetBeans	138
7.4 Le test direct d'un JSP	139
7.5 L'exécution de l'application avec Tomcat standalone	141
7.6 L'utilisation avec Tomcat sur un serveur Linux	143
8. Les déclarations de variables et de méthodes	143
9. L'utilisation des Java Beans	
9.1 Trois actions fondamentales	145
9.2 Un bean pour un JSP	147
9.3 Un JSP qui utilise un bean	150
9.4 Deux JSPs qui utilisent un bean session	152
9.5 Un JSP et un bean avec Tomcat sous Linux	157
10. L'accès aux bases de données par les JSP au moyen des Java Beans	
10.1 Le modèle à 1 bean	158
10.2 Le modèle à 2 beans	163
11. Les pages d'erreurs	167
12. Les balises personnalisées	
12.1 Une classe Java pour une balise sans corps	169
12.2 La classe balise	169
12.3 La création d'un tld	171
12.4 La création du tag handler	173
12.5 La classe SimpleTagSupport	177
12.6 Le JSP utilisateur de la balise	178
12.7 Configuration avec Tomcat comme serveur Web	180
12.8 Le fichier web.xml et les librairies de balises	180
13. Les balises personnalisées avec corps	
13.1 Le JSP attendu	181
13.2 La classe balise avec corps	182
13.3 Les classes associées aux variables de script	184
13.4 La création du tag handler	185
13.5 Le TLD pour une balise personnalisée avec corps	193
13.6 L'exécution du JSP utilisant la balise avec corps	194
14. Les modèles d'architecture à base de JSP	
14.1 Le modèle 1 pour applications simples	198
14.2 Le modèle 2 pour applications complexes	198
15. L'implémentation pratique d'un modèle MVC	
15.1 L'application Web DataCenter	200
15.2 Les données partagées entre JSPs et leur portée	201

15.3 La logique de la servlet contrôleur	202
15.4 Les codes des JSP	203
15.5 Les codes des deux Java beans	206
15.6 Le code la servlet	208
15.7 Un exemple d'utilisation	212

XVIII. Les APIs Java pour XML

1. Les APIs XML	
1.1 Les parsers XML	216
1.2 L'API SAX	217
1.3 L'API DOM	219
2. XML et des packages en plus ...	219
3. Les documents XML sous NetBeans	220
4. Un exemple de parser SAX	
4.1 Le handler de base	225
4.2 L'exception de SAX	226
4.3 Obtenir un parser	226
4.4 Les méthodes du ContentHandler	227
4.5 Le parser minimum	228
5. Un parser qui utilise la réflexion : le principe du projet "robots"	232
6. L'introspection : découvrir les propriétés d'une classe	
6.1 L'API Reflection	234
6.2 S'informer sur la nature d'un objet	234
6.3 S'informer sur les caractéristiques d'une classe	234
6.4 L'instanciation et l'utilisation d'objets	238
6.5 Application aux drivers JDBC	240
7. Un parser qui utilise la réflexion : la réalisation du projet "robots"	
7.1 Les outils	242
7.2 Les classes robots	242
7.3 La méthode factory	244
7.4 L'utilisation de l'API Reflection	247
7.5 L'exécution d'une instruction du script XML	249
7.6 Le programme d'exécution du script robot	249
8. Un exemple de parser DOM	
8.1 Les documents DOM	259
8.2 Obtenir un parser DOM	260
8.3 Obtenir de l'information sur les nœuds	261
8.4 La visualisation du document DOM	263
8.5 Le type des noeuds	265
9. La création directe d'un arbre DOM	
9.1 La création en mémoire	267
9.2 La sauvegarde dans un fichier XML (1)	273
10. Les documents XSL sous NetBeans	275
11. Un exemple de transformation XSLT programmée	
11.1 De nouveaux packages	279
11.2 Obtenir et utiliser un transformer	279
11.3 Configurer le transformer	280

11.4 Un petit programme de transformation	281
11.5 La sauvegarde dans un fichier XML (2)	284
12. La sérialisation XML	
12.1 Une sérialisation texte	286
12.2 Les éléments syntaxiques d'un document XMLEncoder/Decoder	291
12.3 Un exemple démonstratif	292
12.4 La structure d'un document XMLEncoder/Decoder	294

XIX. Un panorama général de J2EE

1. Les quatre plates-formes Java de Sun	296
2. Le principe et l'architecture de RMI	297
3. Un interface d'objet distant	299
4. La classe de l'objet serveur distant	300
5. Un stub et un skeleton	302
6. Le serveur proprement dit	303
7. Un client pour l'objet distant	305
8. Un exemple d'exécution du dialogue RMI	306
9. La nature de l'objet manipulé par le client	307
10. Les objets distants : la suite avec les EJB	
10.1 L'architecture 3-tiers	309
10.2 Les Enterprise Java Beans	310
10.3 Deux types de beans	312
11. Les éléments programmatiques d'un EJB : interfaces et classes (2.0)	
11.1 L'interface distante du composant	312
11.2 L'interface locale de gestion de l'EJB	313
11.3 Une classe bean	314
11.4 Une classe clé primaire	315
11.5 Le container d'EJB	316
11.6 Le descripteur de déploiement	316
11.7 Le client	317
12. Les éléments programmatiques d'un EJB : interfaces et classes (3.0)	
12.1 L'interface distante du composant	312
12.2 La classe bean	314
12.3 Le client	317
13. La plate-forme de développement J2EE	
13.1 Le panorama général de J2EE	318
13.2 Le modèle multi-tiers de J2EE	320

XX. Une introduction aux services Web en Java

1. Un contexte d'informatique distribuée	324
2. Un exemple : le service des provinces	324
3. Le protocole SOAP	325
4. Le langage WSDL	327
5. Le déploiement d'un Web service sous NetBeans	
5.1 La création du Web Service	330
5.2 Le déploiement du Web Service	333

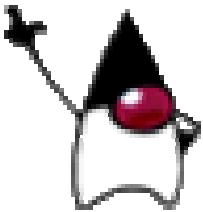
5.3 Le client du Web service	335
6. Le déploiement d'un service selon Apache Axis	
6.1 Le déploiement d'Apache Axis dans Tomcat	338
6.2 Le développement du service Web	339
6.3 Le client du service Web	342

Ouvrages consultés

Annexe : Les bases de XML

1. SGML et XML	A-1
2. Les balises XML	A-2
3. La syntaxe de base d'un document XML	A-3
4. Les parsers et leurs modèles	A-5
5. La validité d'un document XML : la DTD	
5.1 Le contrôle d'un document XML	A-5
5.2 La structure d'une DTD	A-6
6. Un document XML valide pour une DTD	A-8
7. Les attributs dans les DTD	A-10
8. Les entités générales	
8.1 Les entités textuelles	A-13
8.2 Les entités paramètres	A-13
9. Les DTD externes	A-14
10. Les espaces de noms	A-16
11. Les schémas	
11.1 Mieux que les DTDs	A-18
11.2 La définition des éléments de type simple	A-18
11.3 Les types dérivés des types simples	A-19
11.4 La définition des types complexes	A-21
12. XSL et XSLT	
12.1 Le principe	A-23
12.2 La feuille de style selon XSLT	A-23
12.3 Quelques balises xls utiles	A-24
12.4 La transformation d'un fichier XML en page HTML	A-25

Introduction



Dans l'introduction du deuxième volume des notes consacrées à Java, il est écrit que "... le volume 3 présentera la **programmation des applications WEB** ... ". Ce troisième volume de programmation en Java va donc s'attaquer à ce qui constitue sans doute l'un des fers de lance du langage de Sun : la programmation des clients et serveurs WEB sur une plate-forme J2SE (ou même J2EE).

Dans ce contexte, les **servlets**, dont le nom sonne comme "applets" et qui remplacent avantageusement les CGIs écrits en C ou en Perl, vont nous occuper dans un énorme chapitre, cette taille plus que respectable étant justifiée par leurs possibilités étendues. Ce chapitre nous permettra d'ailleurs aussi de faire connaissance avec une référence de fait, soit le serveur **Tomcat**. Et comme si cela ne suffisait pas, ces servlets seront complétées par l'étude des **Java Server Pages**. Bien sûr, le protocole HTTP sera toujours dans les coulisses ...

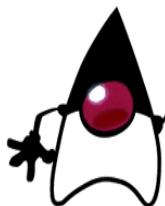
Précisément, en une espèce de suite à HTML, le méta-langage **XML** permet de définir des espèces de protocole de fait de partage de l'information. Et Java ne s'y est pas trompé, puisque ses librairies proposent des outils de traitement comme **JAXP**. Nous y ferons une large promenade.

Enfin, nous terminerons par un panorama général des plates-formes **J2SE** et **J2EE**. Pour cela, nous effleurerons un autre monstre des développements Java : les Enterprise Java Beans, avec le mécanisme **RMI** qu'ils dissimulent. Nous nous permettrons même pour terminer une petite incursion dans le domaine des **services Web**. Mais "effleurer" est le mot qui convient : l'étude convenable de ces concepts, avec d'autres qui leur sont reliés comme JNDI, mériterait bien un autre volume. Tiens, on pourrait l'appeler "programmation distribuée" ;-) ... et confier cela à l'un de mes bons collègues (M.Madani) ;-)

Mais Java nous offre encore bien d'autres possibilités : oui, oui, il y aura un volume IV ;-) ...

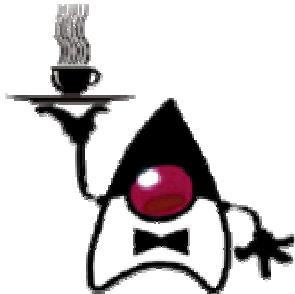
Claude Vilvens

P.S. Les exemples de programmation développés ici l'ont été, selon les cas, en utilisant encore parfois exceptionnellement Sun ONE Studio (ou même Forte for Java) mais surtout **NetBeans 6.***, magnifique environnement de développement. La référence est le JDK 1.5/1.6.



Reprenons donc le fil de nos idées ! Les chapitres sont numérotés en poursuivant la séquence du volume II. Nous commençons donc par le chapitre XVI, qui va nous emmener du côté obscur des serveurs WEB

XVI. Les servlets



On n'a pas encore découvert ce langage qui pourrait exprimer d'un seul coup ce qu'on perçoit en un clin d'œil.

(N. Sarraute, Le Planétarium)

1. L'extension des serveurs

1.1 Le besoin et les solutions non Java

A priori, un serveur WEB ne fournit que des pages HTML, un serveur SMTP des messages, un serveur FTP des fichiers, etc. Mais dans certains cas, et particulièrement celui des serveurs Web, il peut être fort intéressant pour de tels serveurs d'accéder à une base de données, établir une connexion réseau, servir de relais pour transférer une requête vers un autre serveur ou une autre application, etc – autrement dit, utiliser la notion de **middleware** dans une **architecture 3-tiers** (voir chapitre I – volume I). La question est donc d'étendre les possibilités de ces serveurs sans devoir les réécrire ...

Très rapidement, de nombreuses solutions ont été proposées afin d'augmenter les possibilités des serveurs WEBS. Ainsi, on peut citer :

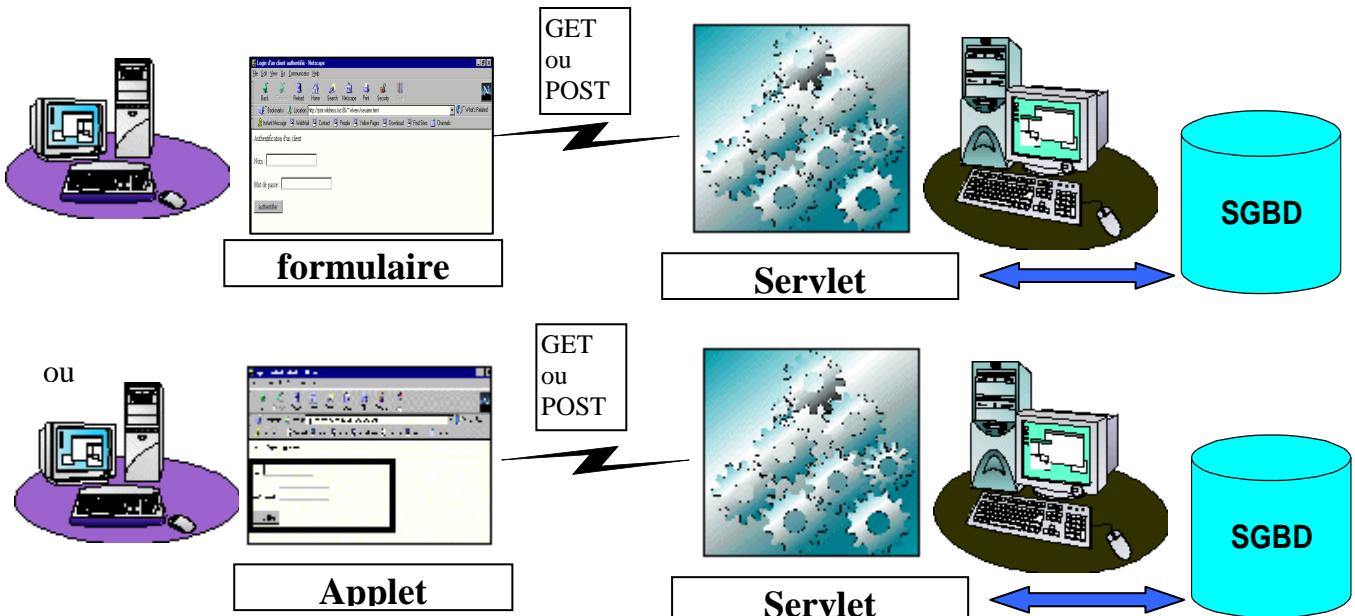
- ◆ les **CGIs** (Common Gateway Interface), programme qui tourne sur le serveur WEB et qui a été lancé à partir d'une requête en provenance d'un interface de navigation client. Un nouveau processus est donc créé pour chaque requête qui parvient au serveur. Les CGIs sont habituellement écrits en C ou en C++ et sont compilés pour fournir un exécutable non portable. A l'heure actuelle, ces CGIs sont aussi souvent écrits en Perl, langage de script aux possibilités assez avancées.
- ◆ les CGIs déclinés en versions particulières, comme **FastCGI** qui crée un seul processus pour des requêtes invoquant le même CGI.
- ◆ les APIs d'extension du serveur : certains constructeurs proposent des APIs capables de modifier les propriétés ou le comportement du serveur qu'ils commercialisent. Ainsi, Netscape propose les **NSAPIs**, Microsoft les **ISAPIs**.
- ◆ les **ASP** (Active Server Pages) de Microsoft : il s'agit de placer du code (le plus souvent ou VBScript ou Jscript) au sein d'une page HTML. Mais ce code n'est pas destiné à être exécuté sur le client mais bien sur le serveur même, avant que la page ne soit envoyée à ce client.
- ◆ les **SSJS** (server-side Javascripts) de Netscape : comme ASP, le code contenu dans les pages génère un contenu dynamique.
- ◆ et, bien sûr, les **Servlets** Java, avec leurs cousines que sont les **JSP** (Java Server Pages) ...

1.2 Les servlets

Dans le même esprit d'extension des serveurs WEBS, et en bref, on peut dire que

les **servlets** sont des unités de code (typiquement, des classes) Java qui étendent les possibilités des serveurs (essentiellement, des serveurs WEB),

Le *modèle 3-tiers* a donc l'aspect classique suivant en programmation Java :



On conçoit sans peine que les servlets peuvent avantageusement remplacer les **CGIs**. En effet, elles¹ peuvent réaliser les mêmes opérations que ceux-ci. Mais, de plus,

- ◆ elles sont écrites en Java et sont exécutées sur une machine virtuelle se trouvant sur le serveur, assurant ainsi la **portabilité** et l'indépendance par rapport à l'environnement de ce serveur (le client n'a absolument pas besoin d'héberger une machine virtuelle);
- ◆ un objet servlet (car c'en est un) n'est **instancié qu'une fois** puis reste en mémoire; ceci signifie donc qu'il n'y aura pas de nouvelle instanciation dans la suite, contrairement aux CGIs qui doivent être recréés (code compilé) ou réinterprétés (scripts) à chaque demande; par conséquent aussi, une servlet est susceptible de conserver des données (comme un compteur ou une connexion à une base de données, par exemple);
- ◆ les servlets sont gérées par des threads au sein du serveur, ce qui leur confère un degré d'intimité avec ce serveur qu'un CGI, processus séparé, ne peut atteindre; de plus, ceci explique pourquoi **une servlet n'est instanciée qu'une seule fois**, même si plusieurs clients la sollicitent simultanément : dans ce cas, **chaque demande client génère un thread et tous ces threads travaillent sur le même objet**; dans ces conditions, on conçoit sans peine l'intérêt des méthodes synchronized pour le code sensible – la servlet devient donc alors un **moniteur**;
- ◆ les mécanismes de sécurité Java peuvent être utilisés.

¹ certains disent "le" servlet – le féminin est employé ici par analogie avec le mot féminin "applet"

Remarque

1) Les serveurs concernés par la technologie logicielle des servlets peuvent être de natures diverses. Cependant, en pratique, on les rencontre principalement comme complément des serveurs utilisant le protocole HTTP.

2) On a coutume de dire que "les servlets sont aux serveurs ce que les applets sont aux browsers". Ce que l'on veut dire par là, c'est que les **servlets** étendent les possibilités des serveurs tout comme les applets permettent d'étendre les opérations possibles au sein d'une page HTML téléchargée par un client. Il y a cependant une énorme différence : l'applet est "*client-side*" alors que la servlet est "*server-side*". De plus, cette dernière n'a pas d'interface graphique, au contraire de l'applet. Enfin, il n'y a pas pour une servlet de restrictions analogues à celles qui frappent les applets.

1.3 Les moteurs à servlets

Un **moteur à servlet** est une application permettant le test et le déploiement des servlets. On parle encore aussi de **container de servlets**. Son rôle est de fournir l'environnement d'exécution des servlets

- ◆ en réalisant la communication entre le serveur Web et les servlets,
- ◆ en gérant le chargement et l'instanciation des servlets ainsi que les invocations de leurs méthodes.

On peut distinguer

- ◆ les moteurs autonomes, qui intègrent directement le support des servlets au sein du serveur WEB; c'est évidemment le plus direct, mais cela lie à la version de ce serveur WEB; des exemples sont Enterprise Server de Netscape ou Domino Go Webserver de Lotus;
- ◆ les moteurs additionnels : ces moteurs ajoutent le support des servlets à un serveur Web qui ne les supporte pas d'origine (comme Apache et IIS); ces moteurs fonctionnent donc comme des plug-in; les plus connus parmi ces moteurs sont le vieux **JServ** (ancien support pour le serveur Web Apache), son successeur **TomCat**, **Jetty** (un Open Source supervisé par Mort Bay Consulting et faisant partie de la licence Apache) ainsi que l'ancien **JRun** de Live SoftWare; tous sont susceptibles de compléter (voire pour certains de remplacer) les serveurs Web les plus courants;
- ◆ les moteurs embarquables : comme leur nom l'indique, il s'agit de serveurs de servlets qui peuvent être embarqués dans une autre application.

2. Les packages pour servlets

Les classes et interfaces nécessaires à l'écriture de servlets se trouvent d'origine dans les JDK récents qui se réclament de **J2SE/J2EE**, essentiellement dans le package **javax.servlet** et ses sous-packages. Celui-ci comporte, globalement :

- ◆ un interface **Servlet** déclarant les méthodes de gestion des servlets et de leurs communications avec les clients;
- ◆ deux interfaces **ServletRequest** et **ServletResponse** correspondant respectivement au traitement d'une requête et à l'envoie d'une réponse;
- ◆ une classe abstraite **GenericServlet** : avec un nom pareil, inutile de préciser que cette classe implémente, notamment, l'interface Servlet; les développeurs de servlets dériveront

- donc leurs classes de celle-ci (ou de sa classe fille HttpServlet); cette classe est indépendante du protocole utilisé;
- ◆ deux classes abstraites flux **ServletInputStream** et **ServletOutputStream**, fournissant une communication depuis et vers le client.

Le sous-package **javax.servlet.http** comporte, comme son nom l'indique, les classes plus spécialement conçues pour le schéma requête/réponse selon HTTP :

- ◆ une classe abstraite **HttpServlet** vouée à l'utilisation des méthodes (au sens de HTPP !) GET, POST, etc;
- ◆ deux interfaces **HttpServletRequest** et **HttpServletResponse** dérivés respectivement de ServletRequest et ServletResponse;
- ◆ une classe utilitaire **HttpUtils**, dont les méthodes sont essentiellement des méthodes de classe et permettent de traiter le résultat d'un POST, de décomposer la QUERY STRING ou encore d'obtenir l'URL du client;
- ◆ et même une classe **Cookie**; un cookie est un moyen pour un serveur d'envoyer de l'information à un client en lui demandant de la mémoriser de manière permanente – ceci permettra au serveur de retrouver ultérieurement ces informations à partir du client – inutile de dire que la sécurité est menacée par un tel procédé.

Nous évoquerons un autre sous-package, javax.servlet.jsp, dans le chapitre consacré aux JSP.

3. **Le cycle de vie d'une servlet**

Une servlet est une classe Java. Elle possède donc un cycle de vie qui rappelle celui d'une applet, mais en plus simple puisqu'il n'y a pas ici de page HTML hôte. Le cycle se ramène ici à 3 points :

3.1 **L'initialisation**

Le bytecode de la servlet doit être chargé en mémoire à l'initiative du serveur (probablement WEB) hôte. A cette occasion, le serveur exécute alors la méthode

```
public abstract void init(ServletConfig config) throws ServletException
```

L'objet exception cité instancie une classe dérivée d'Exception et signale évidemment un problème si il est lancé. Le rôle de l'objet implémentant l'interface **ServletConfig** est, comme on s'en aussi serait douté, de contenir les informations de configuration de la servlet lorsque celle-ci est chargée, mais aussi de fournir une référence vers son container (objet **ServletContext**). Classiquement, on se contente, pour cet aspect des choses, d'appeler la méthode de la super-classe qui se charge de cette initialisation :

```
super.init(config);
```

Mais la méthode init() ne se limite pas à cela. Elle peut, par exemple, mettre en place une connexion JDBC vers une base de données, celle-ci devant être utilisée pour satisfaire les requêtes des clients. Il convient encore de remarquer que la méthode init() n'est donc exécutée qu'une seule fois, au chargement – un serveur ne pourra recharger une servlet que si celle-ci a été au préalable déchargée au moyen de la méthode destroy().

La plupart des serveurs WEB classiques actuels rechargent automatiquement une servlet si son fichier class a été modifié depuis la dernière requête. Cette manière de faire est

propre aux servlets (une classe normale n'est chargée qu'une fois) et est due au fait que *les servlets disposent de leur propre chargeur de classe* (un objet **ClassLoader**); celui-ci agit pour les classes se trouvant dans un répertoire au nom et à la position prédefinis (par exemple, `\servlet(s)` ou **WEB-INF/classes**) – voilà pourquoi les servlets ne peuvent se trouver n'importe où sur le serveur.

3.2 Le traitement des requêtes des clients

La méthode

```
public abstract void service(ServletRequest req, ServletResponse res)
    throws ServletException, IOException
```

a pour tâche de matérialiser le mécanisme requête/réponse en acceptant comme arguments deux objets représentant une requête et une réponse. Nous y reviendrons avec des méthodes plus dédiées à nos problèmes immédiats, soit des requêtes HTTP de type GET et POST.

3.3 La terminaison d'une servlet

Une servlet peut être déchargée de la mémoire par appel de la méthode :

```
public abstract void destroy()
```

qui, en gros réalise le travail inverse de la méthode init(), donc, par exemple, fermer la connexion JDBC vers la base de données utilisée par la servlet. A remarquer que le serveur devrait, sauf cas d'urgence impératif, s'assurer que les services en cours sont terminés ou, du moins, admettre un temps de latence suffisant. Il peut être utile, pour cela, de compter les requêtes en cours de traitement et d'en tenir compte dans la méthode destroy() redéfinie.

4. Le code d'une servlet élémentaire

Imaginons donc vouloir écrire une servlet simple, qui fournira pour toute réponse au client qui la sollicite une page dynamique avec une modeste chaîne de caractères. On crée le code de cette servlet en tant qu'application sans GUI, par exemple dans un répertoire d:`\java-application\TestServlet` :

TestServlet.java

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class TestServlet extends HttpServlet
{
    public void doGet (HttpServletRequest request, HttpServletResponse response)
        throws IOException
    {
        PrintWriter out;
        String titre = "Essai de servlet";
        response.setContentType("text/html");
        out = response.getWriter();
        out.println("<HTML><HEAD><TITLE>");
```

```
out.println(titre);
    // le titre est ce qui est effectivement susceptible d'être dynamique
    // il suffirait qu'il provienne d'une base de données
    out.println("</TITLE></HEAD><BODY>");
    out.println("<H1>" + titre + "</H1>");
    out.println("<p>Réponse de la servlet</p>");
    out.println("</BODY></HTML>");
    out.close();
}
}
```

Comme prévu, notre gentille servlet est une classe héritée de HttpServlet. Elle ne redéfinit pas de constructeur – ce n'est pas nécessaire et, d'ailleurs, le constructeur de la classe mère ne fait rien ... Notre classe redéfinit uniquement la méthode :

```
protected void doGet(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, IOException
```

qui permet de réaliser les opérations répondant à une méthode HTTP GET (méthode par défaut en HTTP). Cette méthode reçoit comme paramètre un objet request (qui encapsule les données issues du client) et un objet response (qui encapsulera la réponse envoyée à ce client); ils vont donc nous permettre de prendre en mains le dialogue requête/réponse. Ainsi, l'objet HttpServletResponse possède la méthode :

```
public abstract void setContentType(String type)
```

qui va permettre d'indiquer le type MIME de la réponse du serveur. Ce même objet HttpServletResponse fournit deux moyens de renvoyer des données vers le client en obtenant un canal, c'est-à-dire un flux, vers celui-ci :

- ◆ public abstract PrintWriter **getWriter()** throws IOException

qui renvoie un objet Writer permettant d'envoyer au client des données de type texte;

- ◆ public abstract ServletOutputStream **getOutputStream()** throws IOException

qui fournit un objet ServletOutputStream qui permettra l'envoi de données sous format binaire. En fait, il s'agit simplement d'un flux java dérivé d'OutputStream et spécialement dédié aux servlets. Ses méthodes se résument à des versions polymorphes de print et println pour tous les types primitifs usuels.

Dans les deux cas,

- ◆ il faut d'abord fixer le header MIME de la réponse;
- ◆ la fermeture du flux indique la fin de la réponse du serveur au client.

Après compilation, on obtient ainsi un fichier bytecode TestServlet.class qui, en situation réelle, doit être placé sur le serveur. La localisation exacte dépend du Web Server et du moteur à servlets utilisés. Dans le cas d'un JavaWebServer, la servlet doit être placée, tout simplement, dans le répertoire servlet. Dans le cas d'un serveur WEB Netscape, elle doit

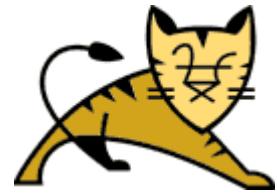
se trouver dans /netscape/suitespot/plugins/java/servlets et pour un serveur WEB Apache dans le répertoire /servlet – mais tout ceci peut se reconfigurer. Une fois placée au bon endroit sur le serveur, on pourra appeler cette servlet au moyen d'un browser et obtenir en réponse la page Web que la servlet aura créée dynamiquement, comme si il s'agissait d'une page statique.

Mais concrètement ? A l'heure actuelle, *le moteur à servlets le plus utilisé est sans doute Tomcat*. Et on en trouve un tout intégré dans NetBeans (et aussi le vieux Sun ONE Studio) ... Nous allons donc faire plus ample connaissance avec lui.

5. Le moteur à servlets Tomcat

5.1 Un moteur à servlets doublé d'un serveur HTTP

Nous utiliserons pour développer nos exemples le moteur à servlets et JSP² dénommé **Tomcat**. Celui-ci est développé par Sun dans le cadre de son projet Jakarta³. Il implémente, en ses versions 5.0/6.0, l'API 2.4/2.5 des servlets et l'API 2.0/2.1 des JSP, et il instancie aussi un container Web (donc de servlets) nommé Catalina. Tomcat⁴ peut fonctionner



- ◆ de manière autonome (en "**standalone**"); il joue alors le rôle d'un serveur HTTP;
- ◆ de manière plus classique, en étant ajouté ("**plugué**") à un serveur Web frontal, comme Apache.

L'intérêt de cette dernière manière de faire (typiquement, *Tomcat plugué dans Apache*) réside dans plusieurs facteurs :

- ◆ la sécurité : le serveur Web frontal subit les assauts du monde extérieur, le container Web reste en arrière, isolé de l'Internet;
- ◆ les performances : Tomcat a pour rôle fondamental d'invoquer les composants Web dynamiques (et suit donc un mode opératoire relativement compliqué) alors qu'un serveur Web a pour rôle de fournir des ressources statiques (dont l'accès est beaucoup plus simple);
- ◆ les possibilités de configuration : un serveur Web comme Apache permet de gérer beaucoup mieux la communication HTTP.

La version standalone est celle que nous allons pourtant privilégier ici, pour notre seule facilité, tant pour Windows que pour Linux/Unix. Sous Windows, le troisième acteur de la trilogie classique d'un serveur WEB utilisant les servlets sera un SGBD (MS-Access ou MySQL sous Windows, MySQL sous Linux – voire Oracle dans les deux cas).

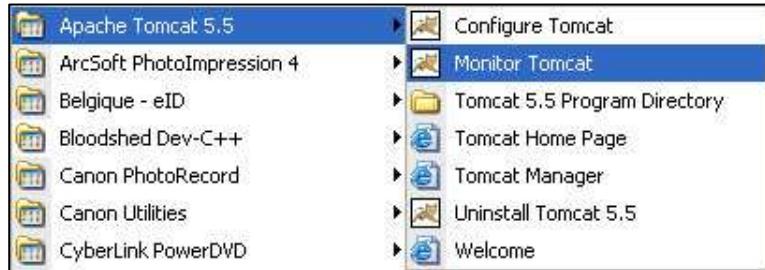
5.2 Tomcat en tant que serveur Web autonome (Windows)

On peut se procurer le logiciel sur le site Web <http://jakarta.apache.org/tomcat>. L'installation est relativement simple, spécialement sous Windows. En fait, la seule chose importante est de spécifier le port d'écoute du serveur (par défaut, c'est un port comme 8080 ou 8081 mais on peut utiliser un port moins évident, comme, par exemple, 8090). Tomcat se lance en tant que serveur indépendant (non intégré) en choisissant dans le menu Démarrer de Windows (il n'y a pas de différence à ce niveau entre les versions 5.5 ou 6) :

² disons provisoirement que les Java Server Pages (JSP) sont des extensions des servlets – voir le chapitre "Les Java Server Pages" pour en savoir plus ...

³ <http://jakarta.apache.org>

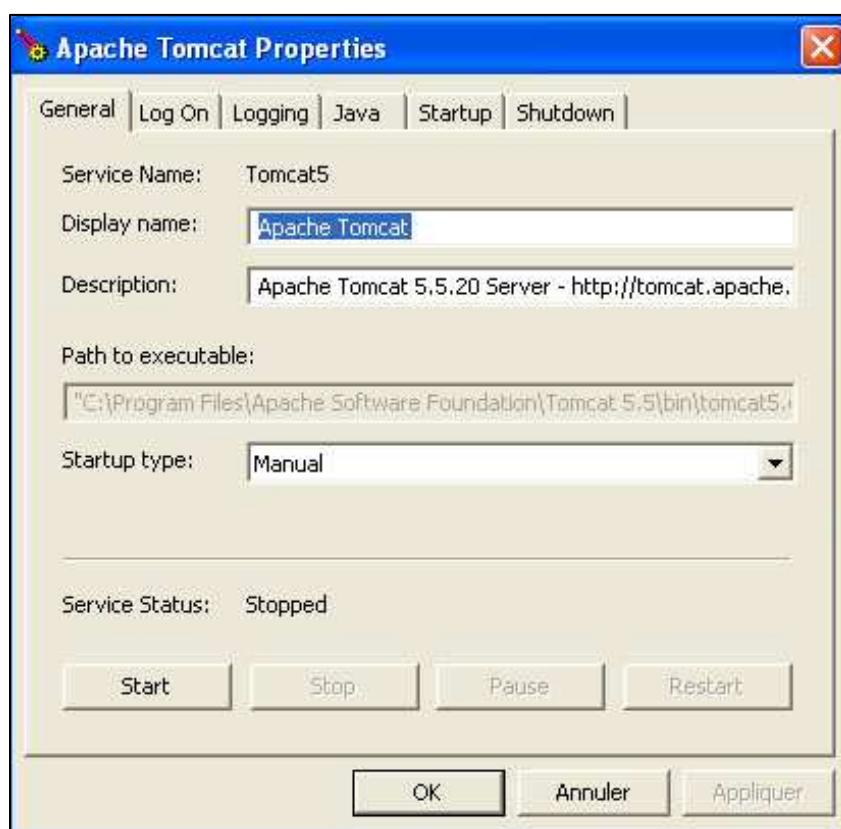
⁴ <http://jakarta.apache.org/tomcat>



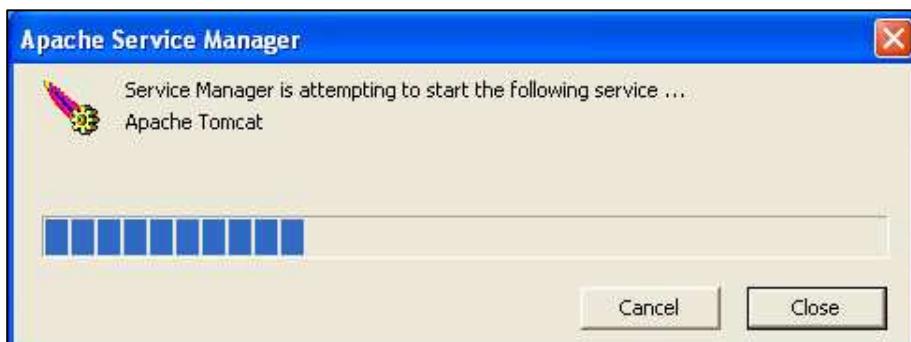
En fait, ceci ne lance pas le serveur mais a pour résultat discret une icône dans la barre d'état :



Un double clic sur cette icône fournit le panneau de monitoring de Tomcat :



On peut donc faire effectivement démarrer Tomcat :



On peut vérifier que le serveur attend sur le port 8080 (ou autre, du moment que ce port n'est pas pris par un autre serveur, comme Apache, IIS ou le Tomcat intégré de NetBeans) :

```
C:\Documents and Settings\ Claude>netstat -an | find "80"
TCP 0.0.0.0:8009      0.0.0.0:0      LISTENING
TCP 0.0.0.0:8080      0.0.0.0:0      LISTENING
TCP 127.0.0.1:8005    0.0.0.0:0      LISTENING
```

On constate que notre serveur attend en fait aussi sur deux autres ports :

- ♦ 8005 : c'est le port de shutdown (donc une espèce particulière de port d'administration);
- ♦ 8009 : est le port du "connecteur JK", donc en fait le port utilisé par un module fonctionnant comme l'ancien moteur à servlets JServ (à pluguer dans Apache), lequel utilisait le protocole AJP (Apache JServ Protocol); c'est évidemment un moyen élégant d'intégrer une ancienne manière de faire et de favoriser la compatibilité.

Bien entendu, on arrête le serveur au moyen du bouton Stop de l'interface de monotoring. Plutôt que de l'arrêter, invoquons plutôt le serveur par browser interposé :

```
http://127.0.0.1:8080/
```

Cela donne :

The screenshot shows a Mozilla Firefox window displaying the Apache Tomcat 5.5.20 default homepage. The address bar shows "http://127.0.0.1:8080/". The page features the Apache Software Foundation logo and a congratulatory message: "If you're seeing this page via a web browser, it means you've setup Tomcat successfully. Congratulations!". On the left, there are three navigation menus: "Administration" (Status, Tomcat Administration, Tomcat Manager), "Documentation" (Release Notes, Change Log, Tomcat Documentation), and "Tomcat Online" (Home Page, FAQ, Bug Database, Open Bugs, Users Mailing List, Developers Mailing List, IRC). At the bottom of the page, the URL "http://java.sun.com/products/servlet" is visible.

On remarque immédiatement les liens importants :

- ◆ la page d'état du serveur :

The screenshot shows the Apache Manager interface running in Mozilla Firefox. The URL in the address bar is `http://127.0.0.1:8080/manager/status`. The page has a header with the Apache Software Foundation logo and a yellow cat icon.

Manager

List Applications	HTML Manager Help	Manager Help	Etat complet du serveur
-------------------	-------------------	--------------	-------------------------

Serveur

Version de serveur	Version de la JVM	Fournisseur de la JVM	Nom d'OS	Version d'OS	Architecture d'OS
Apache Tomcat/5.5.20	1.5.0_06-b05	Sun Microsystems Inc.	Windows XP	5.1	x86

JVM

Free memory: 2.57 MB Total memory: 9.16 MB Max memory: 63.56 MB

http-8080

Max threads: 150 Min spare threads: 25 Max spare threads: 75 Current thread count: 25 Current thread busy: 2
Max processing time: 1828 ms Processing time: 2.796 s Request count: 28 Error count: 4 Bytes received: 0.00 MB Bytes sent: 0.09 MB

Stage	Time	B Sent	B Recv	Client	VHost	Request
R	?	?	?	?	?	?
S	31 ms	0 KB	0 KB	127.0.0.1	127.0.0.1	GET /manager/status HTTP/1.1
R	?	?	?	?	?	?

P: Parse and prepare request S: Service F: Finishing R: Ready K: Keepalive

jk-8009

Max threads: 200 Min spare threads: 4 Max spare threads: 50 Current thread count: 4 Current thread busy: 1
Max processing time: 0 ms Processing time: 0.0 s Request count: 0 Error count: 0 Bytes received: 0.00 MB Bytes sent: 0.00 MB

Stage	Time	B Sent	B Recv	Client	VHost	Request
-------	------	--------	--------	--------	-------	---------

P: Parse and prepare request S: Service F: Finishing R: Ready K: Keepalive

Copyright © 1999-2005, Apache Software Foundation

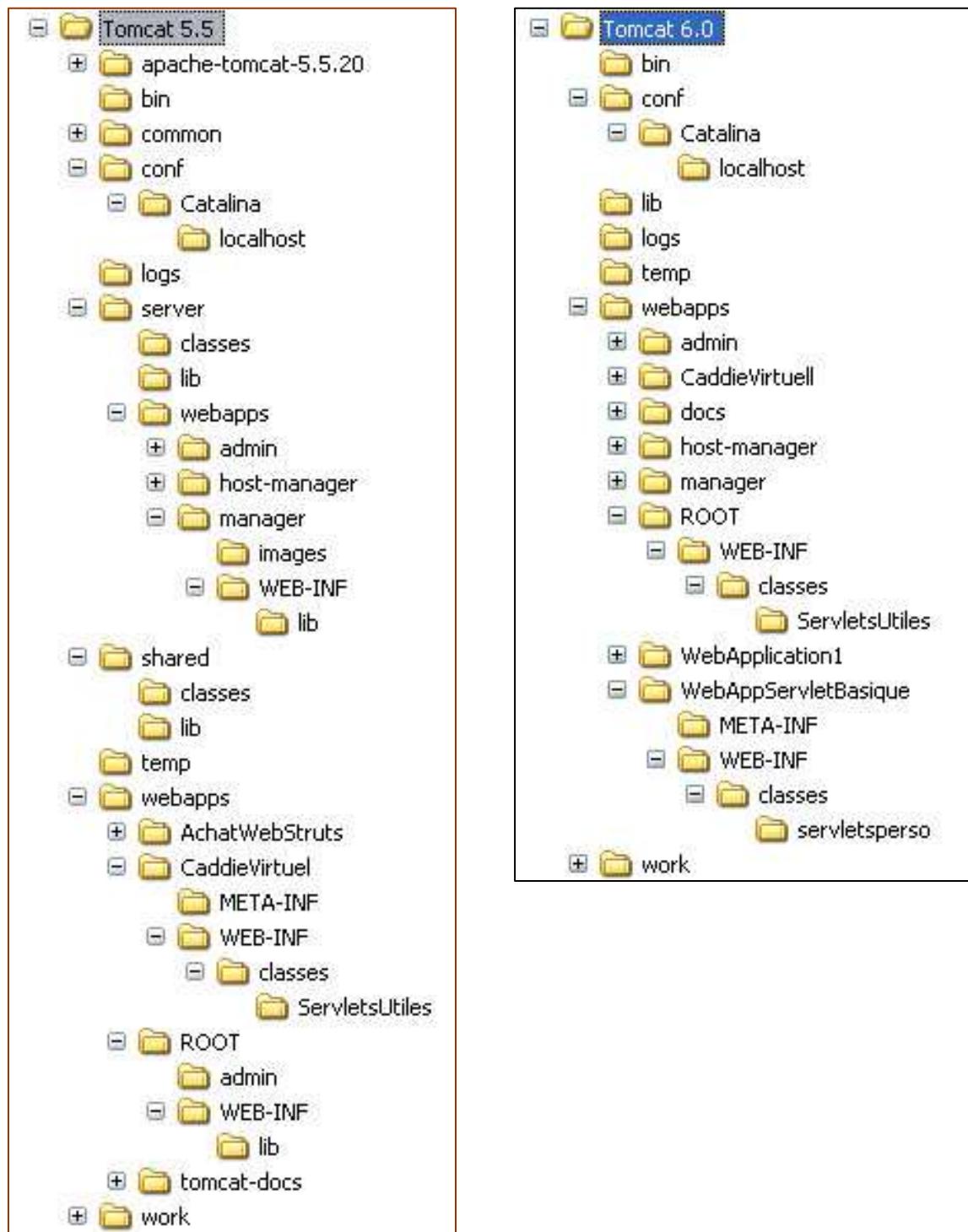
- ◆ la page du manager :

The screenshot shows a Mozilla Firefox browser window. At the top, the address bar displays "http://127.0.0.1:8080/manager/html". The main content area shows the Apache Software Foundation logo and the title "Gestionnaire d'applications WEB Tomcat". Below this, there is a message box containing "Message: OK". A yellow header bar labeled "Manager" contains links for "List Applications", "HTML Manager Help", "Manager Help", and "Etat du serveur". The main content area is titled "Applications" and lists several applications with their URLs, names, status, session counts, and command buttons (Démarrer, Arrêter, Recharger, Undeploy). The applications listed are: / (Welcome to Tomcat), /AchatWebStruts, /CaddieVirtuel, /admin (Tomcat Administration Application), /host-manager (Tomcat Manager Application), /manager (Tomcat Manager Application), and /tomcat-docs (Tomcat Documentation).

- ♦ la page d'administration (pour Tomcat 4 ou 5, si le module est installé - nous en reparlerons) :

The screenshot shows a Mozilla Firefox browser window with the title "Tomcat Server Administration - Mozilla Firefox". The address bar shows "http://127.0.0.1:8080/admin/". The main content area features a large title "TOMCAT WEB SERVER ADMINISTRATION TOOL" in a stylized font. Below the title are fields for "User Name" and "Password", each with a corresponding input box. At the bottom of the form are "Login" and "Reset" buttons. A status message "Terminé" is visible at the bottom left of the page.

On peut aussi vérifier que les répertoires de Tomcat se structurent de la manière suivante :



Cette arborescence trouve sa racine définie dans la variable d'environnement **CATALINA_HOME** dont la valeur est fixée classiquement à l'installation à :

C:\Program Files\Apache Software Foundation\Tomcat 5.5 (ou 6.0)

Nous aurons bien sûr l'occasion d'évoquer certains de ces répertoires.

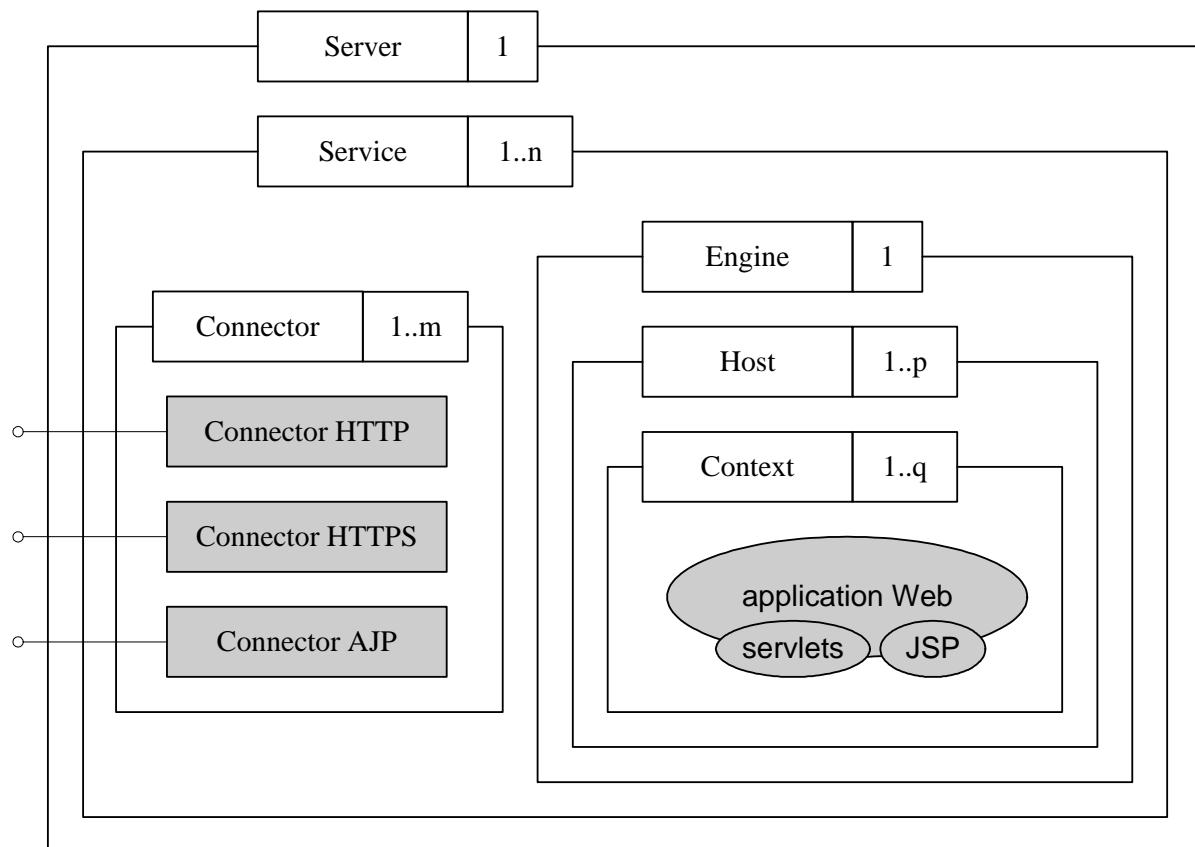
5.3 L'architecture de Tomcat

En fait, il n'est pas vraiment possible d'utiliser un tel serveur sans en connaître un minimum son architecture.

Tomcat est constitué d'un ensemble de composants dont le rôle est de traiter les requêtes envoyées par des clients utilisant le protocole HTTP et souhaitant obtenir telle ou telle ressource. Les composants principaux vont en fait en contenir d'autres et ils appelés "containers" pour cette raison. Leurs noms sont : Server, Service, Engine, Host et Context. Ces composants sont décrits très précisément dans fichier de configuration nommé **server.xml** situé dans

- ◆ un répertoire du type C:\Program Files\Apache Software Foundation\Tomcat 6.0\conf pour Windows (ou C:\Program Files\netbeans-5.5\enterprise3\apache-tomcat-5.5.17\conf pour le Tomcat 5.5 intégré à l'ancien NetBeans 5.5);
- ◆ un répertoire /var/tomcat5/conf pour Linux.

Il n'est pas vraiment nécessaire de posséder des connaissances approfondies de XML pour comprendre les grandes lignes de ce fichier, au demeurant très bien commenté. Et sa lecture permet de bien appréhender l'architecture globale, que l'on pourrait schématiser comme suit :



Chaque composant correspond en fait à une balise de server.xml.

1) On peut constater que la balise la plus extérieure est <Server>, qui précise essentiellement sur quel port Tomcat attend un shutdown; cette balise n'a qu'une occurrence :

<Server port="8005" shutdown="SHUTDOWN">

L'attribut shutdown a en fait pour valeur la chaîne de caractères qui doit être reçue par le serveur pour qu'il se termine effectivement.

2) La balise <Server> est susceptible de contenir un ou plusieurs "**services**", identifiés par une balise <Service>, qui comportent chacun un ou plusieurs connecteurs (balise <Connector>) qui utilisent le même moteur (*engine*). Ainsi, pour le service qui utilisera le moteur à servlet Catalina de Tomcat :

```
<!-- Define the Tomcat Stand-Alone Service -->
<Service name="Catalina">
```

3) Par "**connecteur**", on entend un point d'entrée, caractérisé par un port réseau, sur lequel une requête peut être reçue et par lequel une réponse peut être envoyée; en fait, pour ce qui nous intéresse ici, la requête sera transmise à une servlet tournant dans le container associé au connecteur. Par exemple, le connecteur correspondant au protocole HTTP courant est :

```
<!-- Define a non-SSL HTTP/1.1 Connector on port 8080 -->
<Connector port="8080" maxHttpHeaderSize="8192"
           maxThreads="150" minSpareThreads="25" maxSpareThreads="75"
           enableLookups="false" redirectPort="8443" acceptCount="100"
           connectionTimeout="20000" disableUploadTimeout="true" />
```

si 8080 est le port spécifié à l'installation pour Tomcat en tant que serveur Web. On remarquera dans les attributs toute une série de paramètres concernant **le pool de threads** lancé par Tomcat au démarrage, notamment le nombre maximum et le nombre minimum de threads inactifs qui permet à Tomcat d'ajuster le nombre de threads du pool ☺.

Un autre connecteur correspond à une connexion par HTTPS :

```
<!-- Define a SSL HTTP/1.1 Connector on port 8443 -->
<!--
<Connector port="8443" maxHttpHeaderSize="8192"
           maxThreads="150" minSpareThreads="25" maxSpareThreads="75"
           enableLookups="false" disableUploadTimeout="true"
           acceptCount="100" scheme="https" secure="true"
           clientAuth="false" sslProtocol="TLS" />
-->
```

4) On peut définir des "**engines**", qui constituent des points d'entrée pour les requêtes HTTP parvenue par les connecteurs et qui déterminent l'hôte concerné par cette requête. Ainsi, le moteur à servlet Catalina de Tomcat correspond à l'*engine* :

```
<Engine name="Catalina" defaultHost="localhost">
```

A remarquer que le service et l'*engine* portent par défaut le même nom, ce qui n'est pas très intéressant pour les écritures dans le fichier de log : le mieux est de modifier le nom du service (par exemple "Tomcat-standalone") pour faciliter la lecture de ce fichier de log en cas de nécessité.

5) En fait, un engine est capable de référencer des "**hôtes**" divers, comme la machine locale ou un hôte virtuel (technique qui permet de faire tourner plusieurs sites webs pour la même adresse IP). On peut le vérifier par la présence, à l'intérieur d'un engine, de balises comme :

```
<Host name="localhost" appBase="webapps" unpackWARs="true" autoDeploy="true"  
      xmlValidation="false" xmlNamespaceAware="false">
```

On remarquera ainsi l'attribut appBase, qui permet de définir le répertoire racine à partir duquel sont stockées les applications Web accessibles par cet hôte. Ici, ce répertoire est donc **CATALINA_HOME/webapps**.

6) Le point essentiel, du simple point de vue du développement, est cependant la balise "**contexte**" que l'on trouve au sein d'une balise Host. Cette balise <Context> représente en effet tout simplement la description d'un application Web; dans la version la plus simple, sa syntaxe est :

```
<Context path=...  
        docBase=...  
        debug=...  
        reloadable=...>  
</Context>
```

Ses attributs préciseront essentiellement, par exemple pour chaque utilisateur (Linux) ou chaque projet (NetBeans) :

- ◆ le répertoire de base de l'arborescence définie ci-dessus (attribut **docBase**);
- ◆ l'alias associé à ce répertoire et qui sera utilisé par Tomcat (attribut **path**).

En pratique, Tomcat crée un élément <Context> en mémoire pour toutes les applications qui sont présentes dans le "répertoire de publication des applications", c'est-à-dire celui qui est précisé dans l'attribut appBase de la balise Host. C'est le **contexte par défaut**. Donc, si on déploie toujours les applications Web dans ce répertoire, il n'y aura aucun contexte à décrire ☺.

Si on ne souhaite pas pratiquer ainsi, on pourra donc définir une balise <Context> dans server.xml. Mais il existe encore un autre moyen : définir des fichiers **context.xml** – nous en verrons un exemple avec le module d'administration de Tomcat. Le contexte par défaut est mis en service avec le fichier context.xml qui se trouve dans CATALINA_HOME/conf :

conf\context.xml

```
<!-- The contents of this file will be loaded for each web application -->  
<Context>  
    <!-- Default set of monitored resources -->  
    <WatchedResource>WEB-INF/web.xml</WatchedResource>  
    <!-- Uncomment this to disable session persistence across Tomcat restarts -->  
    <!--  
    <Manager pathname="" />  
    -->  
</Context>
```

En résumé, un squelette de base de server.xml pour la version 5.5 (en 6.0, le contenu est encore plus simple) est celui-ci :

conf\server.xml (schéma général)

```
<Server port="8005" shutdown="SHUTDOWN">
    <Service name="Catalina">
        <Connector port="8080" maxHttpHeaderSize="8192"
            maxThreads="150" minSpareThreads="25" maxSpareThreads="75"
            enableLookups="false" redirectPort="8443" acceptCount="100"
            connectionTimeout="20000" disableUploadTimeout="true" />
        <Connector port="8443" maxHttpHeaderSize="8192"
            maxThreads="150" minSpareThreads="25" maxSpareThreads="75"
            enableLookups="false" disableUploadTimeout="true"
            acceptCount="100" scheme="https" secure="true"
            clientAuth="false" sslProtocol="TLS" />
        <Connector port="8009"
            enableLookups="false" redirectPort="8443" protocol="AJP/1.3" />
    <Engine name="Catalina" defaultHost="localhost">
        <Host name="localhost" appBase="webapps"
            unpackWARs="true" autoDeploy="true"
            xmlValidation="false" xmlNamespaceAware="false">
            <Context path="/examples" docBase="examples"
                reloadable="true" crossContext="true">
                <Logger className="org.apache.catalina.logger.FileLogger"
                    prefix="localhost_examples_log." suffix=".txt" timestamp="true"/>
            </Context>
        </Host>
    </Engine>
</Service>
</Server>
```

5.4 Tomcat en tant que serveur Web autonome (Linux)

Dans le cas de Tomcat installé sur une machine Linux, le fichier server.xml contient une clause *<Context ...>* pour chaque utilisateur qui spécifie un path différent pour chacun d'entre eux. On a donc quelque chose du genre :

server.xml (Linux)

```
<Server port="8005" shutdown="SHUTDOWN" debug="0">
    <Service name="Tomcat-Standalone">
        <Connector className="org.apache.catalina.connector.http.HttpConnector"
            port="8180" minProcessors="5" maxProcessors="75"
            enableLookups="true" redirectPort="8543"
            acceptCount="10" debug="0" connectionTimeout="60000"/>
        <Engine name="Standalone" defaultHost="localhost" debug="0">
            <Host name="localhost" debug="0" appBase="webapps" unpackWARs="true">
                ...
                <!-- Contextes pour Deneumos -->
                <Context path="/LibrairieTag"
                    docBase="/stage/deneumos/LibrairieTag2" />
            </Host>
        </Engine>
    </Service>
</Server>
```

```

        debug="0" reloadable="true">
    <Logger className="org.apache.catalina.logger.FileLogger"
    prefix="localhost_pw_log." suffix=".txt"
    timestamp="true"/>
</Context>

<Context    path="/testJDBC"
    docBase="/stage/deneumos/testJDBC"
    debug="0" reloadable="true">
    <Logger className="org.apache.catalina.logger.FileLogger"
    prefix="localhost_pw_log." suffix=".txt"
    timestamp="true"/>
</Context>
..
<Context    path="/webdev"
    docBase="/stage/deneumos/webdev"
    debug="0" reloadable="true">
    <Logger className="org.apache.catalina.logger.FileLogger"
    prefix="localhost_pw_log." suffix=".txt"
    timestamp="true"/>
</Context>

<!--Contexte pour utilisateur Vilvens -->
<Context    path="/webdev2"
    docBase="/home/vilvens/webdev"
    debug="0" reloadable="true">
    <Logger className="org.apache.catalina.logger.FileLogger"
    prefix="localhost_pw_log." suffix=".txt"
    timestamp="true"/>
</Context>
</Host>
</Engine>
</Service>

<Service name="Tomcat-Apache"> ... </Service>
</Server>
```

Chaque utilisateur va donc en fait utiliser utiliser le path qui lui est propre pour invoquer ces ressources Web – ainsi, l'utilisateur Vilvens utilisera dans ses URLs "/webdev2". Nous y reviendrons.

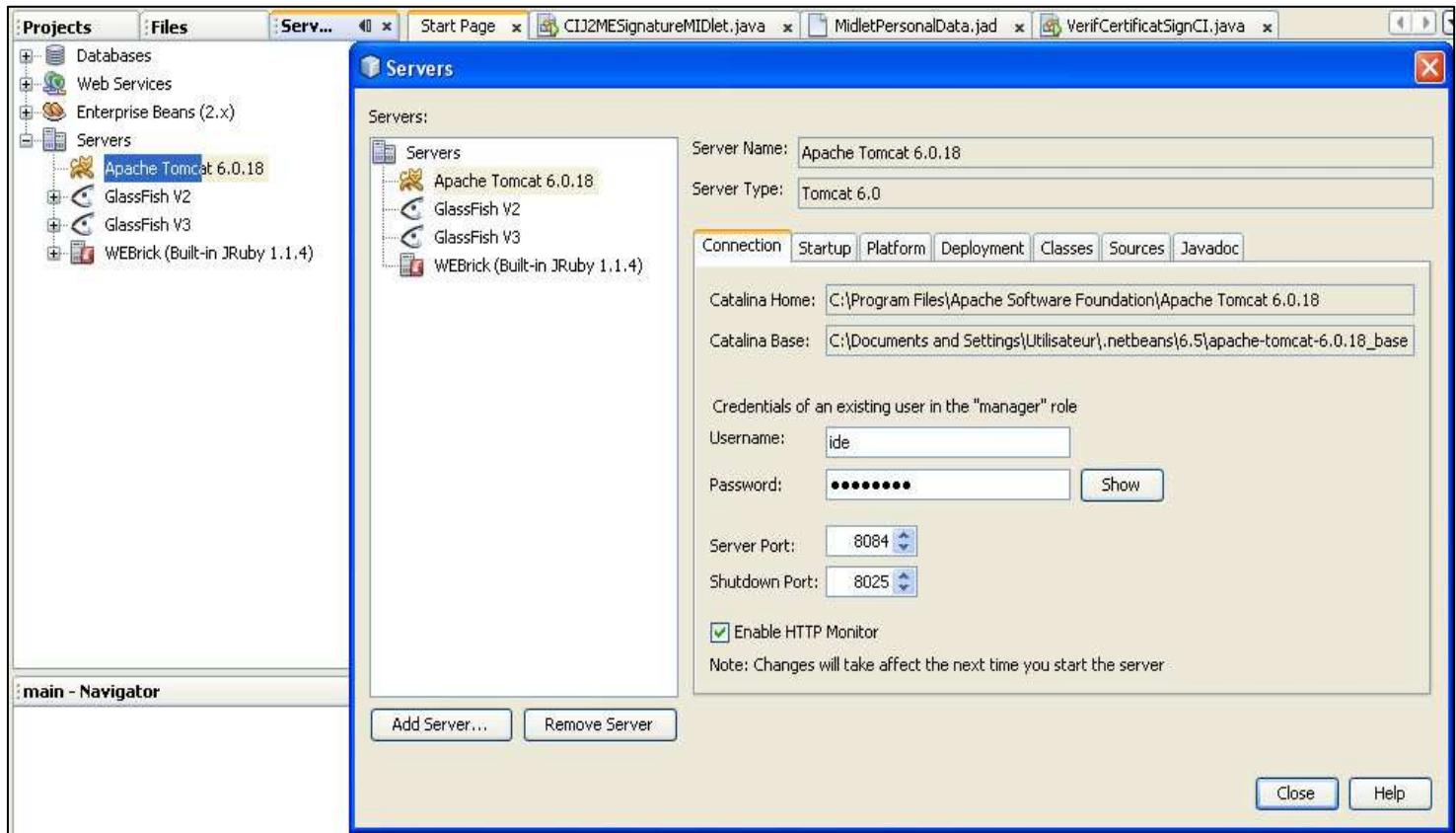
5.5 Tomcat dans NetBeans

La version standalone est également intégrée en sa version 6 au sein de NetBeans 6.*. Tomcat y attend par défaut sur un port fixé à l'installation de NetBeans (par exemple, 8084 - mais on peut modifier ce port d'écoute). Le répertoire de base est cette fois :

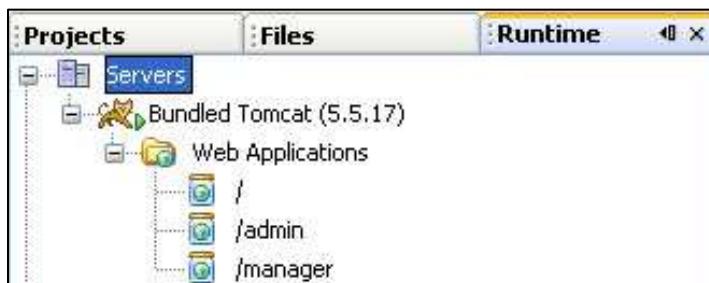
C:\Program Files\Apache Software Foundation\Tomcat 6.0.18\webapps\ROOT

(c'était C:\Program Files\netbeans-5.5\enterprise3\apache-tomcat-5.5.17\webapps\ROOT avec Netbeans 5.5).

On peut visualiser les caractéristiques du serveur dans l'onglet Services de la fenêtre de navigation :



On peut le faire démarrer par le choix Start dans le menu contextuel obtenu par un clic droit sur le nœud Tomcat. On obtient :



Chacune des applications webs repérées peut être démarrée par le choix "Open in Browser" dans le menu contextuel obtenu par un clic droit sur le nœud de l'application.

Mais que l'on utilise Tomcat standalone ou Tomcat intégré à Netbeans, notre servlet doit être transformée en un "composant Web" – qu'est-ce donc ?

5.6 Les applications Web et les Web application ARchive

Par "*application Web*" (ou *Web component* ou encore *Web module*), on entend une application constituant une *extension dynamique d'un serveur Web*; autrement dit, elle est capable de générer des pages Web **dynamiques** contenant des balises HTML ou XML, mais contient aussi des ressources **statiques**, comme des images.

Typiquement, les "**composants Web**" sont des servlets ou des JSP. La plate-forme sur laquelle ces composants sont instanciés pour être rendus utilisables est appelée un "**Web container**" et c'est Tomcat qui sera ce container.

Une application Web est distribuée sous forme d'un **WAR** (Web application ARchive) qui est l'équivalent d'un JAR pour les librairies de classes groupées en package. Typiquement **une application Web, et donc un WAR, comporte**

- ◆ les applications ou composants Web (servlets, JSP);
- ◆ les classe utilitaires utilisées par le serveur, comme les beans d'accès aux bases de données;
- ◆ les éléments statiques (pages HTML, images, sons);
- ◆ les classes clients (applets, classes utilitaires pour ce client).

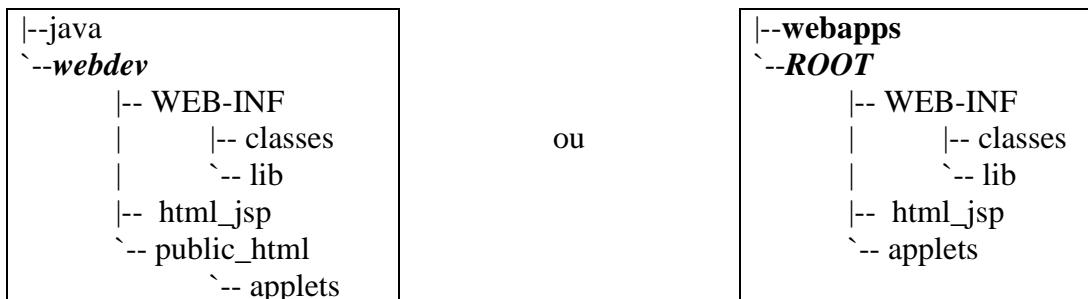
Une application Web, et donc un WAR, doit avoir une certaine structure. En particulier, Tomcat travaille en utilisant l'arborescence de répertoires suivante :

- ◆ un répertoire de base (ou racine), dont le nom (par exemple, **webdev** ou **webapps/ROOT**) est quelconque mais qui devra être précisé dans le fichier de configuration **server.xml**; ce répertoire contiendra les pages JSP, les ressources statiques, les classes et archives destinées aux clients;
- ◆ un sous-répertoire appelé obligatoirement **WEB-INF**, qui contiendra notamment le fichier de configuration **web.xml**, celui-ci, qui permet de spécifier le contenu de l'application Web pour qu'elle soit utilisable, va nous intéresser pour l'utilisation des servlets;
- ◆ deux sous-répertoires de WEB-INF, appelés nécessairement **classes** et **lib**; classes a pour rôle de contenir les servlets (et les *.class Java utilisées par les JSP - des beans par exemple) tandis que lib contient les fichiers *.jar.

Bien sûr, rien n'interdit de définir des sous-répertoires additionnels dans le répertoire racine. Par exemple sous Linux, où tous les développements sont groupés par utilisateur, on peut imaginer des répertoires :

- ◆ **jsp** pour y placer les JSP;
- ◆ **public_html** pour y placer les pages html.

Schématiquement, cela donne donc (respectivement une machine Linux et une machine Windows) :



Bien sûr, nous pourrions confectionner ce war de nos propres mains : mais les environnements de développement permettent d'en obtenir un de manière bien plus simple et plus sûre. Nous allons voir bientôt ce qu'il en est pour Netbeans. Mais auparavant ...

5.7 Le descripteur de déploiement web.xml

Nous y avons déjà fait allusion, le déploiement d'une servlet au sein d'une application Web est décrit par un fichier web.xml, encore appelé le "descripteur de déploiement" de la servlet. Plus précisément, les éléments XML d'un tel descripteur sont définis en détail dans le document de spécification des servlets. Ils doivent se succéder, quand ils sont présents, dans un ordre bien précis. Ne retenons que les clauses les plus courantes :

- 1) les paramètres d'initialisation sont définis dans une zone <context-param> :

```
<context-param>
    <param-name> ... </param-name>
    <param-value> ... </param-value>
</context-param>
```

On peut les récupérer au moyen de la méthode du ServletContext :

```
public ObjectgetAttribute(String name)
```

- 2) la définition d'un classe listener qui gère les événements relatifs au cycle de vie d'une servlet – par exemple, une classe qui implémente un interface **ServletContextListener** correspondant à l'événement d'initialisation de la servlet :

```
<listener>
    <listener-class>...</listener-class>
</listener>
```

- 3) le point fondamental, c'est-à-dire les diverses servlets et leur chemin d'accès :

```
<servlet>
    <servlet-name>...</servlet-name>
    <display-name>...</display-name>
    <description>...</description>
    <servlet-class>... </servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>...</servlet-name>
    <url-pattern>...</url-pattern>
</servlet-mapping>
```

Par exemple, on pourrait avoir :

```
<servlet>
    <servlet-name>FormServlet</servlet-name>
    <servlet-class>ServletsUtiles.FormServlet</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>FormServlet</servlet-name>
    <url-pattern>/FormServlet</url-pattern>
</servlet-mapping>
```

Décryptons ;-):

- ◆ le tag <servlet> : Il existe donc dans l'application une servlet instance de la classe FormServlet (du package ServletsUtilis) : elle sera désignée sous le nom de *FormServlet* (en fait, nous pourrions utiliser n'importe quel nom sans rapport avec la classe Java correspondante).
- ◆ le tag <servlet-mapping> : La servlet connue sous le nom *FormServlet* sera démarrée par l'intermédiaire d'un browser utilisant le protocole HTTP au moyen de l'URL du serveur complétée par "/FormeServlet" (encore une fois, nous pourrions utiliser n'importe quel nom sans rapport avec la classe Java correspondante).

4) le temps maximum de durée d'une session (en minutes – typiquement 30 minutes) :

```
<session-config>
    <session-timeout>
        30
    </session-timeout>
</session-config>
```

5) les pages d'erreurs associées à des classes d'exception :

```
<error-page>
    <exception-type> ... </exception-type>
    <location> ... </location>
</error-page>
```

En pratique, les environnements de développement se chargeront de remplir les clauses nécessaires du fichier web.xml. Le développeur n'aura donc plus qu'à éventuellement modifier l'une ou l'autre ligne selon ses désirs.

Remarque

Tomcat possède en réalité un fichier *web.xml* dans son répertoire conf. Ce descripteur de déploiement est utilisé pour les applications Web qui n'ont pas de web.xml propre. Ainsi, on y trouve le paramètre d'expiration des sessions (30 minutes) :

```
<session-config>
    <session-timeout>30</session-timeout>
</session-config>
```

De plus, ce fichier décrit le déploiement de deux servlets particulières :

a) la "servlet par défaut" : elle fournit les ressources statiques des applications Web comme les pages HTML ou les images :

DefaultServlet dans conf/web.xml

```
<servlet>
    <servlet-name>default</servlet-name>
    <servlet-class>org.apache.catalina.servlets.DefaultServlet</servlet-class>
    <init-param>
        <param-name>debug</param-name>
```

```
<param-value>0</param-value>
</init-param>
<init-param>
    <param-name>listings</param-name>
    <param-value>false</param-value>
</init-param>
<load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>default</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>
```

b) la **servlet des Java Server pages** : c'est elle qui transforme les JSPs en servlets – mais nous en reparlerons beaucoup plus loin :

JspServlet dans conf/web.xml

```
<servlet>
    <servlet-name>jsp</servlet-name>
    <servlet-class>org.apache.jasper.servlet.JspServlet</servlet-class>
    <init-param>
        <param-name>fork</param-name>
        <param-value>false</param-value>
    </init-param>
    <init-param>
        <param-name>xpoweredBy</param-name>
        <param-value>false</param-value>
    </init-param>
    <load-on-startup>3</load-on-startup>
</servlet>

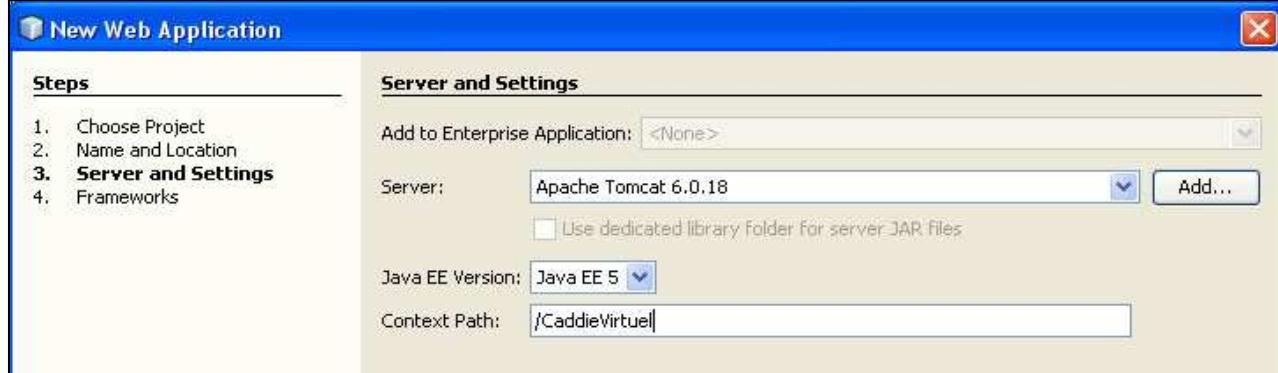
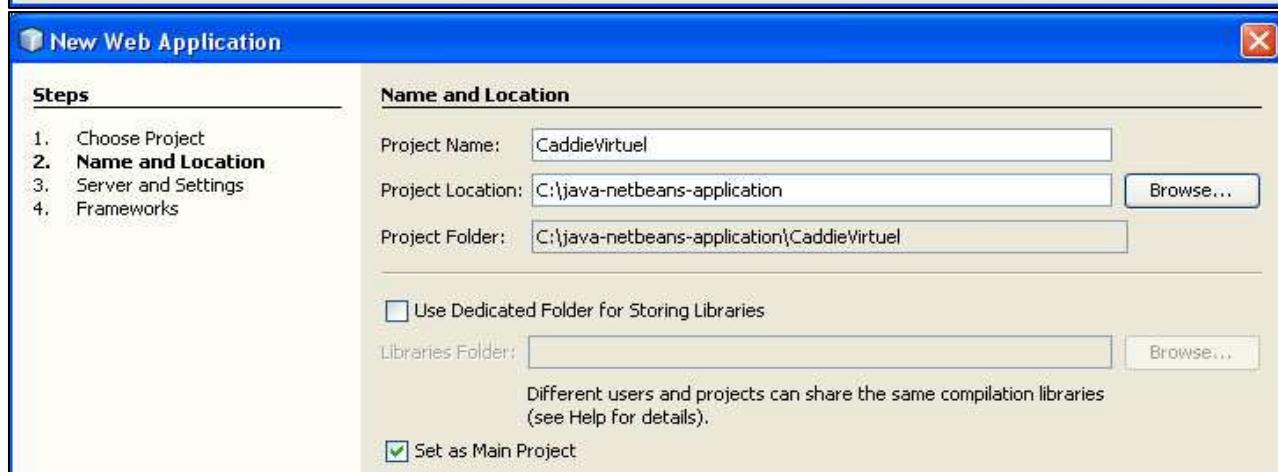
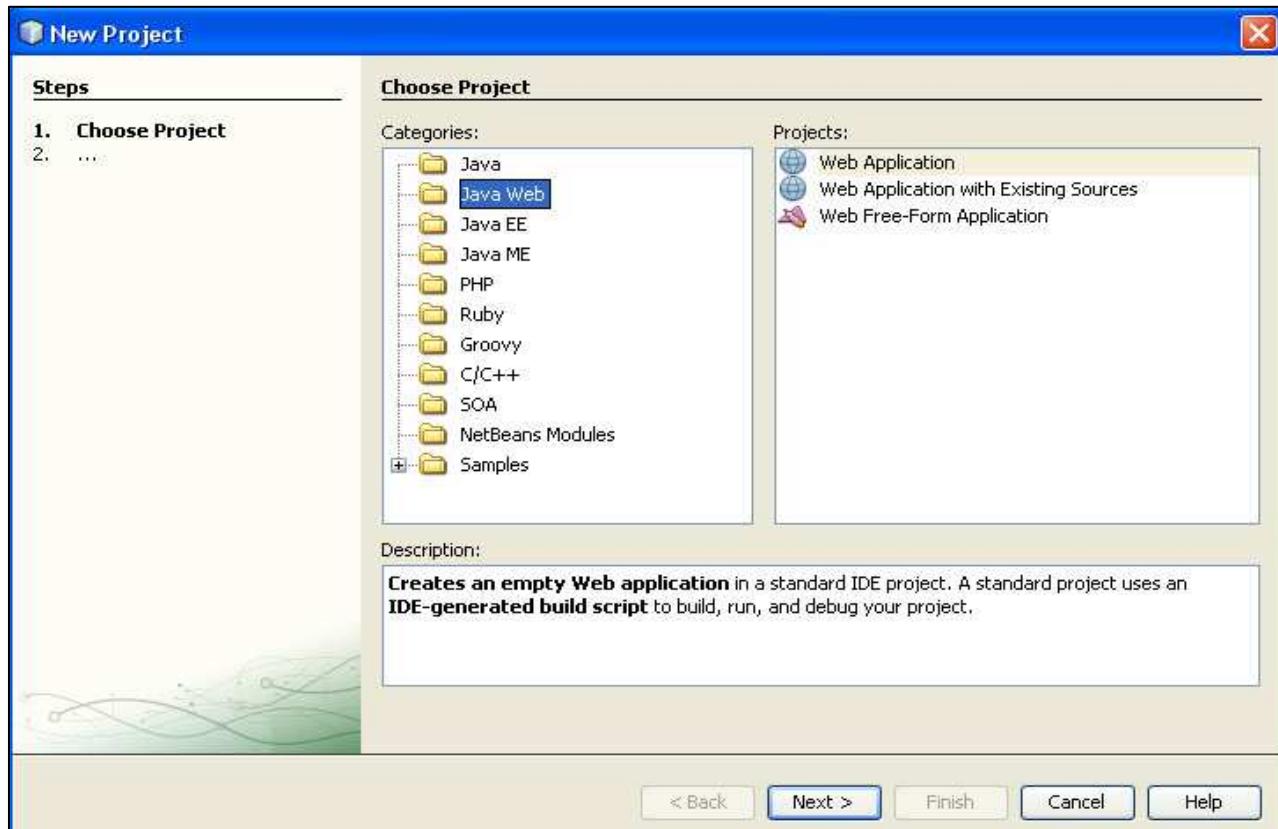
<servlet-mapping>
    <servlet-name>jsp</servlet-name>
    <url-pattern>*.jsp</url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>jsp</servlet-name>
    <url-pattern>*.jspx</url-pattern>
</servlet-mapping>
```

Venons-en maintenant aux choses concrètes ...

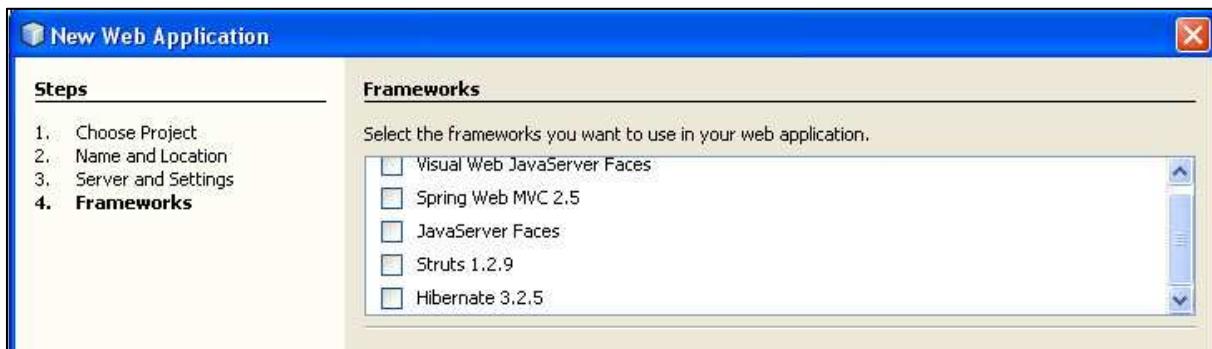
6. Le développement d'une servlet avec NetBeans et Tomcat

6.1 Mise en place du contexte Web

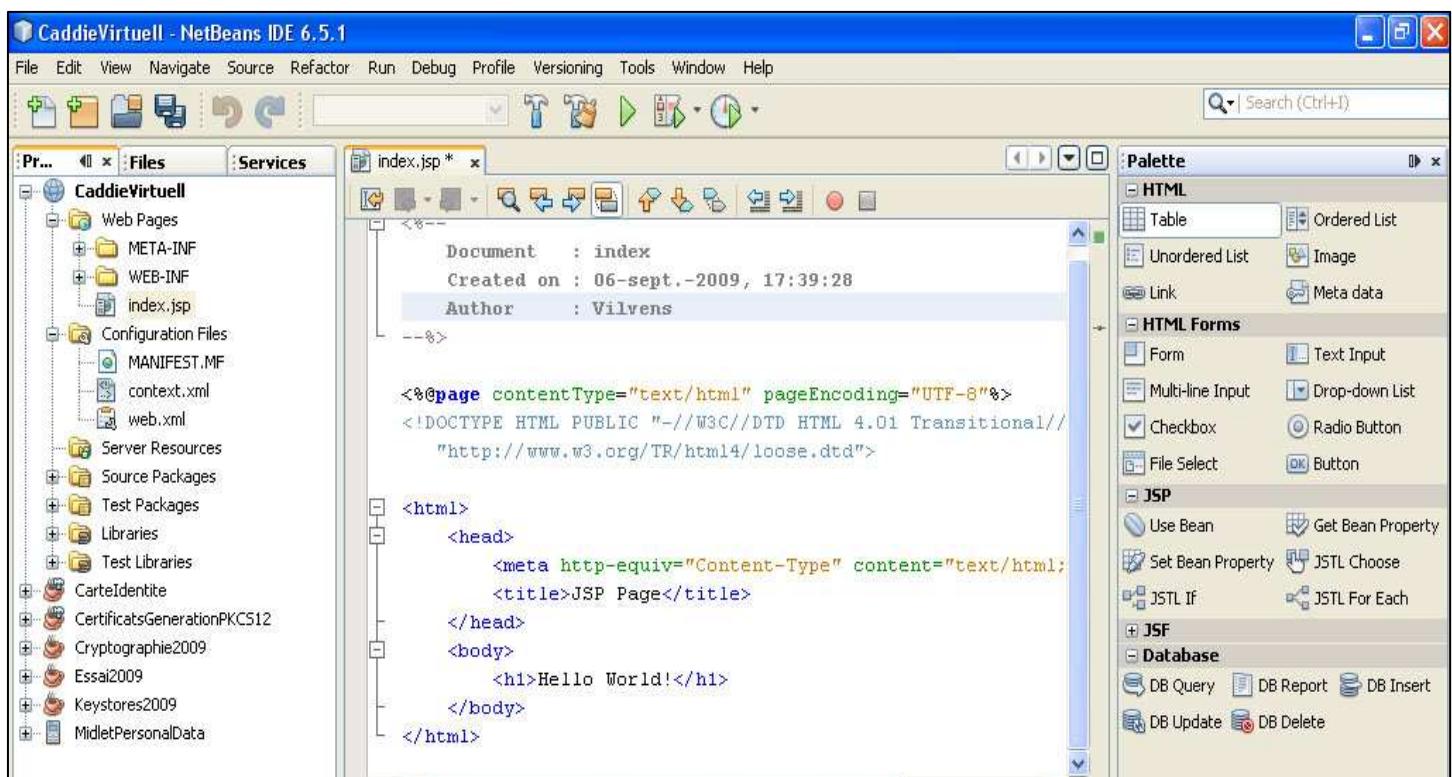
Le projet que nous allons démarrer est donc cette fois une application Web. En démarrant avec File → New Project :



On remarquera, outre le nom du projet ("CaddieVirtuel"), le choix de Tomcat 6 et à la plate-forme J2EE 5. La dernière étape ne nous intéresse guère ici, puisqu'elle demande le framework éventuellement utilisé –ici, nous passons⁵ :



On se retrouve ainsi dans un contexte classique d'application Web :



On remarquera, à droite, la palette des composants Web qui ne demande qu'à être utilisée ...

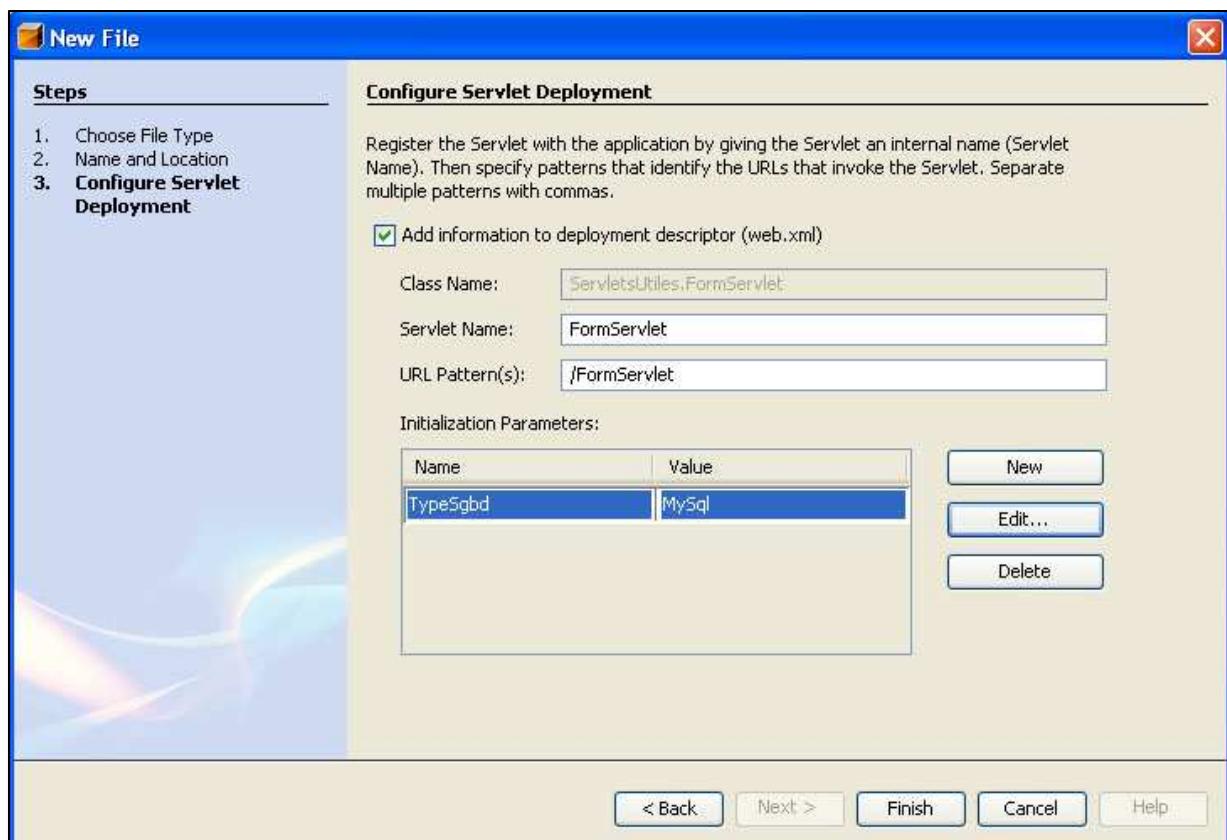
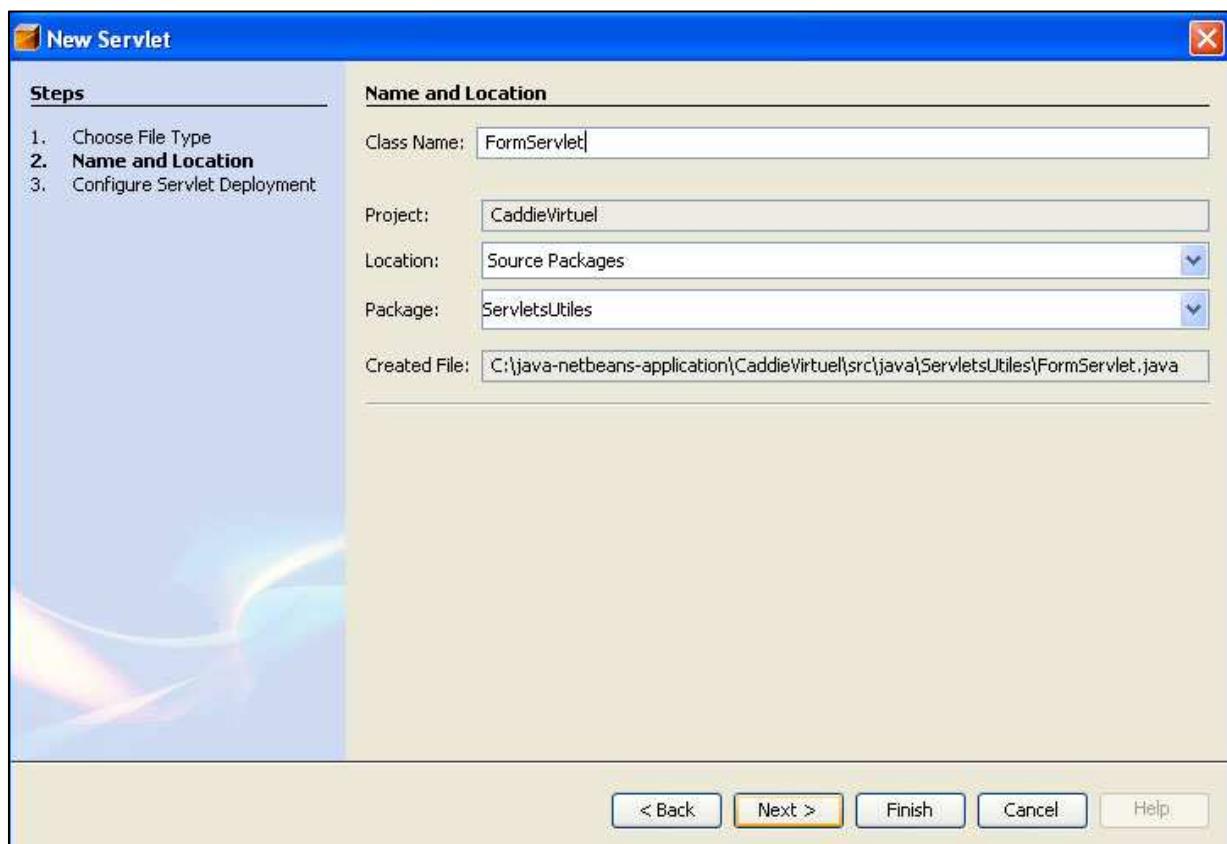
6.2 La création d'une servlet

Une fois l'application Web créée, nous pouvons entamer le développement d'une servlet, nommée par exemple FormServlet : elle traite une entrée classique dans un formulaire. Dans le projet, nous créons tout d'abord un package ServletsUtiles dans les SourcePackages. Un clic droit sur ce package et

New → Servlet

nous conduit à un wizzard assez prévisible :

⁵ mais il n'en sera pas toujours ainsi – voir Struts dans Java IV (du même auteur ...)



On remarquera la définition d'un paramètre d'initialisation (TypeSgbd), possibilité que nous évoquerons plus loin. On obtient ainsi la génération d'une servlet de base :

FormServlet.java (code généré)

```
/*
 * FormServlet.java
 *
 * Created on 23 juin 2006, 17:59
 */

package ServletsUtiles;

import java.io.*;
import java.net.*;

import javax.servlet.*;
import javax.servlet.http.*;

/**
 *
 * @author Vilvens
 * @version
 */
public class FormServlet extends HttpServlet {

    /** Processes requests for both HTTP <code>GET</code> and <code>POST</code>
     * methods.
     * @param request servlet request
     * @param response servlet response
     */
    protected void processRequest (HttpServletRequest request, HttpServletResponse
response)
        throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        /* TODO output your page here
        out.println("<html>");
        out.println("<head>");
        out.println("<title>Servlet FormServlet</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("<h1>Servlet FormServlet at " + request.getContextPath () + "</h1>");
        out.println("</body>");
        out.println("</html>");
        */
        out.close();
    }

    // <editor-fold defaultstate="collapsed" desc="HttpServlet methods. Click on the + sign on
    // the left to edit the code.">
    /** Handles the HTTP <code>GET</code> method.
     * @param request servlet request
     *
```

```

 * @param response servlet response
 */
protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
    processRequest(request, response);
}

/** Handles the HTTP <code>POST</code> method.
 * @param request servlet request
 * @param response servlet response
 */
protected void doPost(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
    processRequest(request, response);
}

/** Returns a short description of the servlet.
 */
public String getServletInfo() {
    return "Short description";
}
// </editor-fold>
}

```

On remarquera, outre l'envoi des tags minimaux pour constituer une page dynamique, la méthode **processRequest()** qui n'est pas une méthode du package standard des servlets, mais qui réalise le traitement effectué par la servlet : son seul intérêt est d'être appelée indifféremment par les méthodes **doGet()** et **doPost()**, mettant ainsi les deux traitements sur le même pied – on est évidemment toujours libre de changer cela. Nous pouvons à présent tailler ce modèle à la mesure de nos désirs :

FormServlet.java (code modifié)

```

/*
 * FormServlet.java
 *
 * Created on 23 juin 2006, 17:59
 */

package ServletsUtiles;

import java.io.*;
import java.net.*;

import javax.servlet.*;
import javax.servlet.http.*;

/**
 * @author Vilvens
 */

```

```
public class FormServlet extends HttpServlet
{
    protected void processRequest(HttpServletRequest request,
                                 HttpServletResponse response) throws ServletException, IOException
    {
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<head>");
        out.println("<title>Servlet de traitement de données</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("<h6>Réponse fournie par la servlet FormServlet at " +
                   request.getContextPath () + "</h6>");
        out.println("<p><h3>Bonjour toi ;-) !</h3><p>");
        out.println("</body>");
        out.println("</html>");
        out.close();
    }
    ...
}
```

Après compilation, le fichier web.xml (dans le WEB-INF des "Web Pages" ou les "Configuration Files" du projet) a été adapté à l'existence de cette servlet :

web.xml

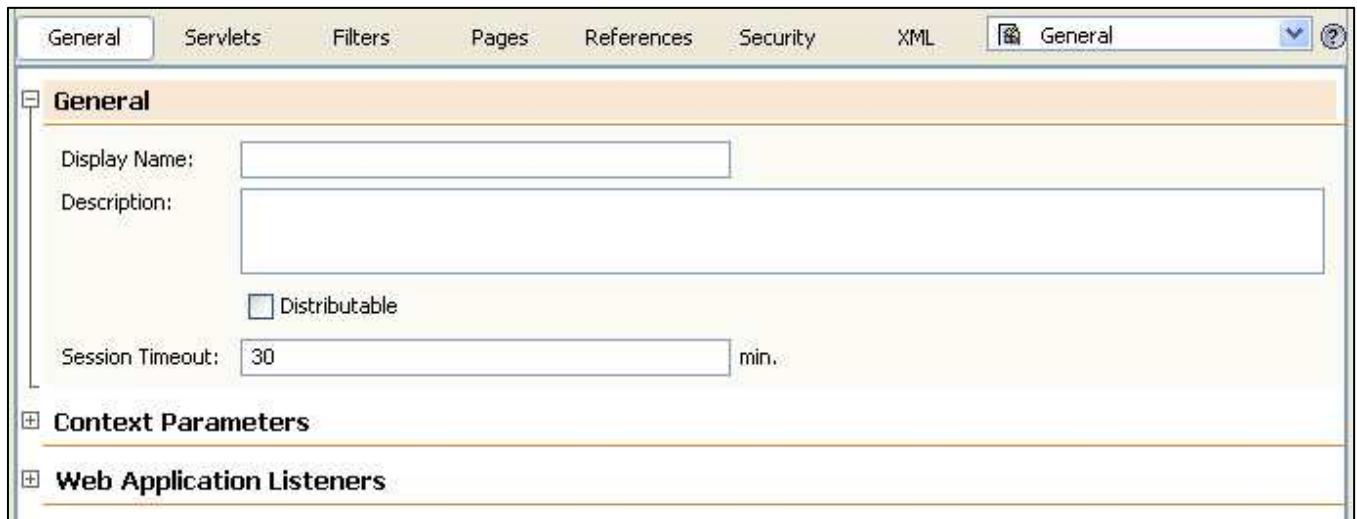
```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://java.sun.com/xml/ns/j2ee/web-
app_2_4.xsd">

    <servlet>
        <servlet-name>FormServlet</servlet-name>
        <servlet-class>ServletsUtiles.FormServlet</servlet-class>
        <init-param>
            <param-name>TypeSgbd</param-name>
            <param-value>MySql</param-value>
        </init-param>
    </servlet>
    <servlet-mapping>
        <servlet-name>FormServlet</servlet-name>
        <url-pattern>/FormServlet</url-pattern>
    </servlet-mapping>
    <session-config>
        <session-timeout>
            30
        </session-timeout>
    </session-config>

```

```
<welcome-file-list>
    <welcome-file>
        index.jsp
    </welcome-file>
</welcome-file-list>
</web-app>
```

On peut aussi visualiser ce fichier web.xml de manière plus "friendly" :



6.3 L'exécution de la servlet

La servlet peut alors être invoquée directement depuis un browser par

<http://localhost:8084/CaddieVirtuel/FormServlet>

tandis que l'exécution de la servlet par l'EDI (Run File) donne d'abord :



pour obtenir finalement :



On pourra examiner avec intérêt dans la fenêtre des consoles l'onglet "HTTP Monitor" :

The first screenshot shows the 'Request' tab of the HTTP Monitor. It displays a list of recorded requests on the left and detailed information about a selected request on the right. The selected request is for '/CaddieVirtuel/FormServlet' via GET. The details include Request URI, Method (GET), Query string (empty), Protocol (HTTP/1.1), Client IP address (127.0.0.1), Scheme (http), and HTTP exit status (Could not be determined). A note states 'There was no query string'.

The second screenshot shows the 'Headers' tab of the HTTP Monitor. It displays a list of recorded requests on the left and detailed HTTP headers for a selected request on the right. The selected request is for '/CaddieVirtuel/FormServlet'. The headers listed include host (localhost:8084), user-agent (Mozilla/5.0 (Windows; U; Windows NT 5.1; fr; rv:1.8.1.4) Gecko/20070515 Firefox/2.0...), accept (text/xml, application/xml, application/xhtml+xml, text/html;q=0.9, text/plain;q=0.8, image...), accept-language (fr,fr-fr;q=0.8,en-us;q=0.5,en;q=0.3), accept-encoding (gzip,deflate), accept charset (ISO-8859-1,utf-8;q=0.7,*;q=0.7), keep-alive (300), and connection (keep-alive).

On pourra encore vérifier que l'application Web a bien été reconnue comme telle dans l'onglet Runtime :



On peut encore créer une page HTML (dans le répertoire Web Pages), ou encore modifier la page index.jsp, qui permettra d'invoquer la servlet :

```
FormPage.html
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">

<html>
<head>
<title></title>
</head>
```

```
<body>
<form action="FormServlet">
    <input type="submit" value="Lancer la servlet" />
</form>
</body>
</html>
```

L'exécution de cette page depuis NetBeans donne :



7. Le déploiement d'une servlet avec Tomcat standalone

Une fois la servlet considérée comme étant au point, comment la déployer sur un serveur Tomcat qui n'est pas intégré à NetBeans ?

7.1 Le déploiement d'une servlet sous forme de fichier class

Dans son installation de base, Tomcat utilise comme répertoire de référence C:\Program Files\Apache Software Foundation\Tomcat 6.0\webapps\ROOT. Il suffit donc à nouveau de créer à partir de ce point une structure d'application Web. On peut constater que le sous-répertoire **WEB-INF** existe déjà et qu'un fichier **web.xml** basique existe :

web.xml (initial)

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<web-app 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"
  version="2.4">

  <display-name>Welcome to Tomcat</display-name>
  <description>
    Welcome to Tomcat
  </description>

  <!-- Tomcat 5.5 seulement
  <servlet>
    <servlet-name>org.apache.jsp.index_jsp</servlet-name>
    <servlet-class>org.apache.jsp.index_jsp</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>org.apache.jsp.index_jsp</servlet-name>
    <url-pattern>/index.jsp</url-pattern>
  </servlet-mapping>
  -->
</web-app>
```

Il suffit donc

- ♦ de créer un sous-répertoire **classes** dans WEB-INF, puis un sous-répertoire **ServletsUtiles** dans celui-ci et d'y placer finalement **FormServlet.class**;
- ♦ de modifier web.xml pour qu'il prenne connaissance de l'existence de la servlet :

web.xml (après modification)

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app
  PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
  "http://java.sun.com/dtd/web-app_2_3.dtd">

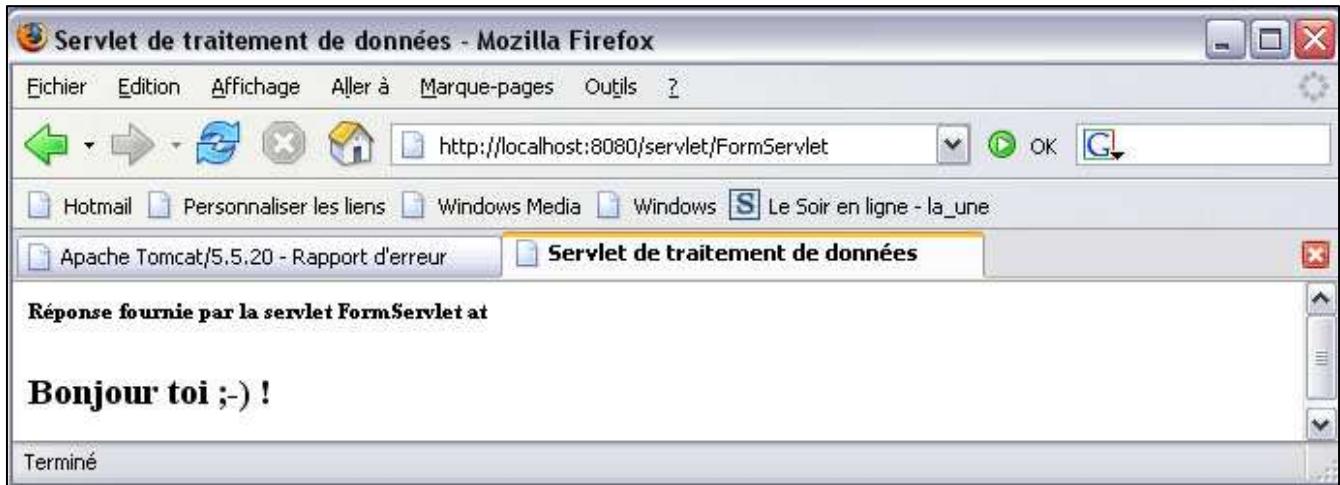
<web-app>
  <display-name>Welcome to Tomcat</display-name>
  <description>
    Welcome to Tomcat
  </description>

  <servlet>
    <servlet-name>FormServlet</servlet-name>
    <servlet-class>ServletsUtiles.FormServlet</servlet-class>
  </servlet>
```

```
<servlet-mapping>
<servlet-name>FormServlet</servlet-name>
<url-pattern>/servlet/FormServlet</url-pattern>
</servlet-mapping>

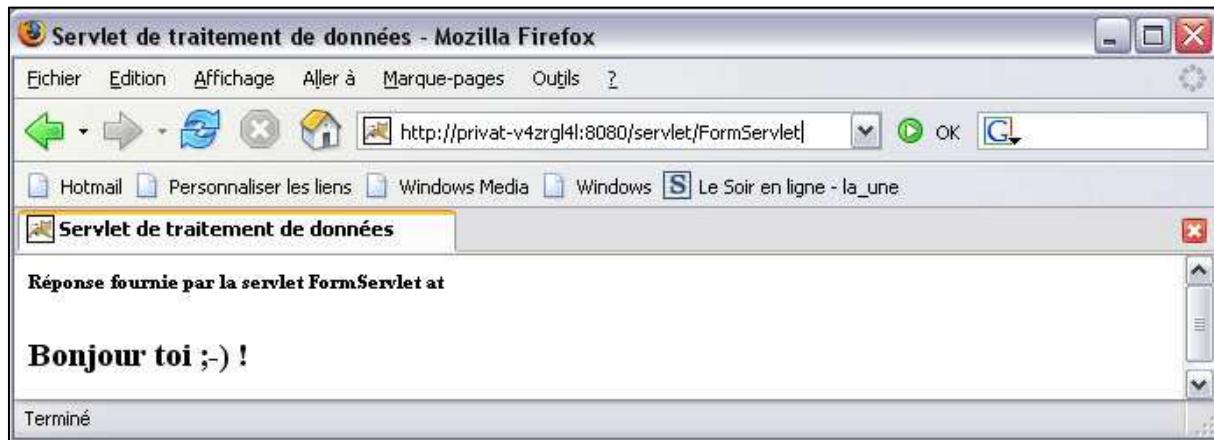
</web-app>
```

Si Tomcat est, par exemple, installé avec 8080 comme port d'écoute, on peut le lancer et invoquer la servlet :



On peut faire de même avec Tomcat sur une machine distante au nom DNS barbare ;-) :

<http://privat-v4zrgl4l:8080/servlet/FormServlet>



7.2 Le déploiement d'une servlet sous forme de WAR avec le contexte par défaut

On peut y constater qu'après compilation, un **war** (CaddieVirtuel.war) a été créé dans le sous-répertoire dist (comme "distribution").



Ce fichier

C:\java-netbeans-application\CaddieVirtuel\dist\ **CaddieVirtuel.war**

contient en fait le fichier **FormServlet.class** dans WEB-INF\classes\ServletsUtiles. Dans l'EDI, la fenêtre projets propose une vision logique dans son onglet Projects. L'onglet Files donne une vision plus claire quant au déploiement.

Pour déployer la servlet, il suffit

- ♦ de copier le WAR dans le répertoire webapps du Tomcat distant qui sera le serveur Web
- ♦ d'appeler la page HTML de démarrage depuis un browser en utilisant le nom du WAR pour y accéder, par exemple ici :

<http://privat-v4zrgl4l:8080/CaddieVirtuel/FormPage.html>

ou

<http://privat-v4zrgl4l:8080/CaddieVirtuel/index.jsp>





Bien sûr, on peut aussi accéder à la servlet directement :

http://privat-v4zrgl4l:8080/CaddieVirtuel/FormServlet

Attention ! *Les choses se passent aussi facilement parce que Tomcat est configuré pour qu'il en soit ainsi.* En effet, dans la balise <Host>, on a positionné à true les attributs :

- ◆ **autoDeploy="true"** : indique que les applications Webs placées dans le répertoire désigné par l'attribut appBase (ici : **appBase="webapps"**) doivent être automatiquement déployées ou pas; pour faciliter cela soit du point de vue développement, cette option est le plus souvent désactivée sur un serveur de production car
 - Tomcat doit surveiller le répertoire en permanence;
 - si un hacker parvient à placer un WAR dans ce répertoire, celle-ci sera automatiquement considérée comme valide !
- ◆ **deployOnStartup="true"** : indique que les nouvelles applications Webs se voient affecter automatiquement un contexte d'application Web par défaut (ce qui les rend donc exécutables);
- ◆ **unpackWARS="true"** : permet la décompression des archives, donc notamment des WARS.

7.3 Le déploiement d'une servlet sous forme de WAR avec un contexte spécifique

Evidemment, on peut ne pas souhaiter déposer dans webapps le WAR matérialisant l'application Web, ne fût-ce que par souci de sécurité. Dans ce cas, il faut définir un contexte explicite pour celle-ci . Ceci peut se faire de deux manières :

- 1) On peut ajouter une balise **Context** dans **server.xml**. Par exemple, si nous voulons désigner notre application CaddieVirtuel développée avec NetBeans sans la déplacer, nous pouvons modifier server.xml de la manière suivante :

dans **server.xml**

```
...
<Host name="localhost" appBase="webapps"
  unpackWARS="true" autoDeploy="true" deployOnStartup="true"
  xmlValidation="false" xmlNamespaceAware="false">
```

```
<Context path="/CaddieVirtuel"  
       docBase="C:\java-netbeans-application\CaddieVirtuel\build\web" />  
...
```

Cette manière de faire était celle utilisée dans le fichier server.xml issu du monde Unix/Linux (point 5.4). L'inconvénient, bien que tout soit ainsi centralisé, est qu'il faut relancer le serveur pour qu'il prenne connaissance de la nouvelle version de son fichier de configuration.

2) On peut aussi placer la définition du contexte dans un fichier XML séparé, fichier qu'il faudra placer dans

CATALINA_HOME/conf/<nom_engine>/<nom_host>

donc pour nous qui n'avons rien changé à la configuration originale :

C:\Program Files\Apache Software Foundation\Tomcat 5.5\conf\Catalina\localhost.

Encore une fois, il est sous-entendu que l'attribut autoDeploy de la balise <Host> est à true.

7.4 Le déploiement du module d'administration de Tomcat 5

On procède de cette manière pour installer le module d'administration de Tomcat 5, qui n'est pas installé automatiquement (c'était le cas dans la version 4 de Tomcat et il n'existe plus pour Tomcat 6). Après avoir téléchargé le fichier zip d'administration, on dispose notamment de ce fichier :

admin.xml

```
<!-- Context configuration file for the Tomcat Administration Web App  
 $Id: admin.xml 303123 2004-08-26 17:03:35Z remm $  
-->  
<Context docBase="${catalina.home}/server/webapps/admin" privileged="true"  
antiResourceLocking="false" antiJARLocking="false">  
- <!--  
Uncomment this Valve to limit access to the Admin app to localhost for obvious security  
reasons. Allow may be a comma-separated list of hosts (or even regular expressions).  
<Valve className="org.apache.catalina.valves.RemoteAddrValve"  
allow="127.0.0.1"/>  
-->  
</Context>
```

Ce fichier est donc à copier dans :

C:\Program Files\Apache Software Foundation\Tomcat 5.5\conf\Catalina\localhost

Il reste donc à copier l'application elle-même, qui est le répertoire admin du /server/webapps du zip, dans le répertoire correspondant de Tomcat 5.5, soit

C:\Program Files\Apache Software Foundation\Tomcat 5.5\server\webapps

On peut alors vérifier que l'outil d'administration est à présent disponible (il faut parfois redémarrer la machine ...).

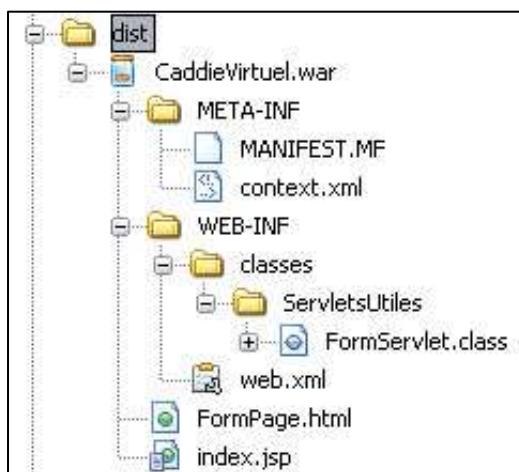
7.5 Le déploiement d'une servlet sous forme de WAR selon le couple Tomcat-NetBeans

Tomcat permet en fait de placer le fichier XML de contexte au sein même du WAR, dans le répertoire META-INF – et c'est ce que fait NetBeans. Dans ce cas, le fichier se limite à définir le path :

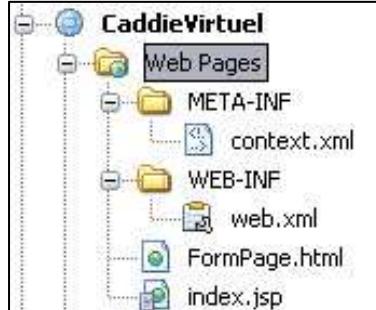
context.xml dans CaddieVirtuel.war

```
<?xml version="1.0" encoding="UTF-8"?>
<Context path="/CaddieVirtuel" />
<!-- Tomcat 6 : Context antiJARLocking="true" path="/CaddieVirtuel" -->
```

Le contenu précis du War est visible dans l'onglet Files :



On peut voir aussi l'existence d'un fichier index.jsp (déjà évoqué), également d'ailleurs dans l'onglet Projects :



Il s'agit en fait d'un Java Server Page (sujet du chapitre suivant) qui, à ce stade, n'est rien de bien plus qu'une page HTML et qui sert implicitement de page de départ pour l'application Web – nous l'avons modifiée légèrement (comme déjà évoqué précédemment) :

index.jsp (modifié)

```
<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>

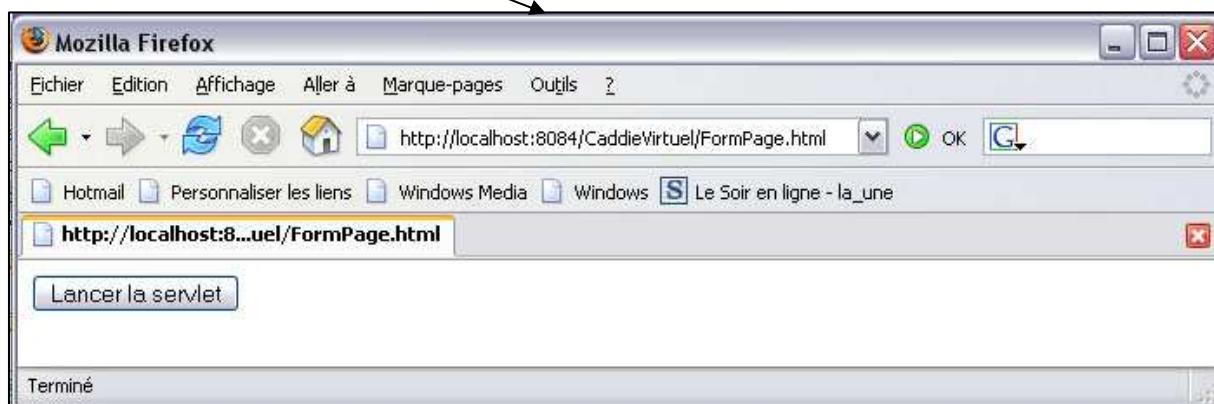
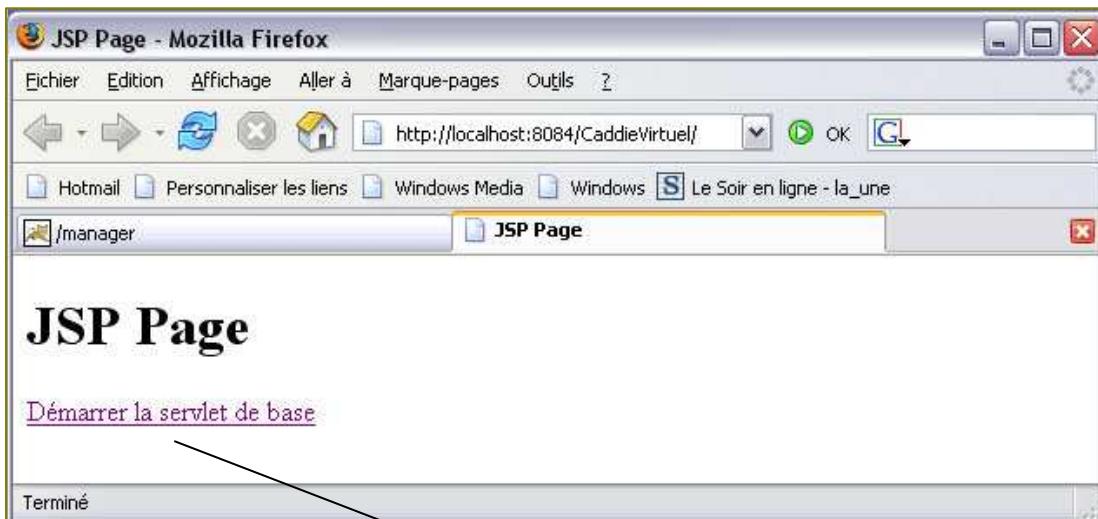
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
 "http://www.w3.org/TR/html4/loose.dtd">

<html>
```

```
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>JSP Page</title>
</head>
<body>

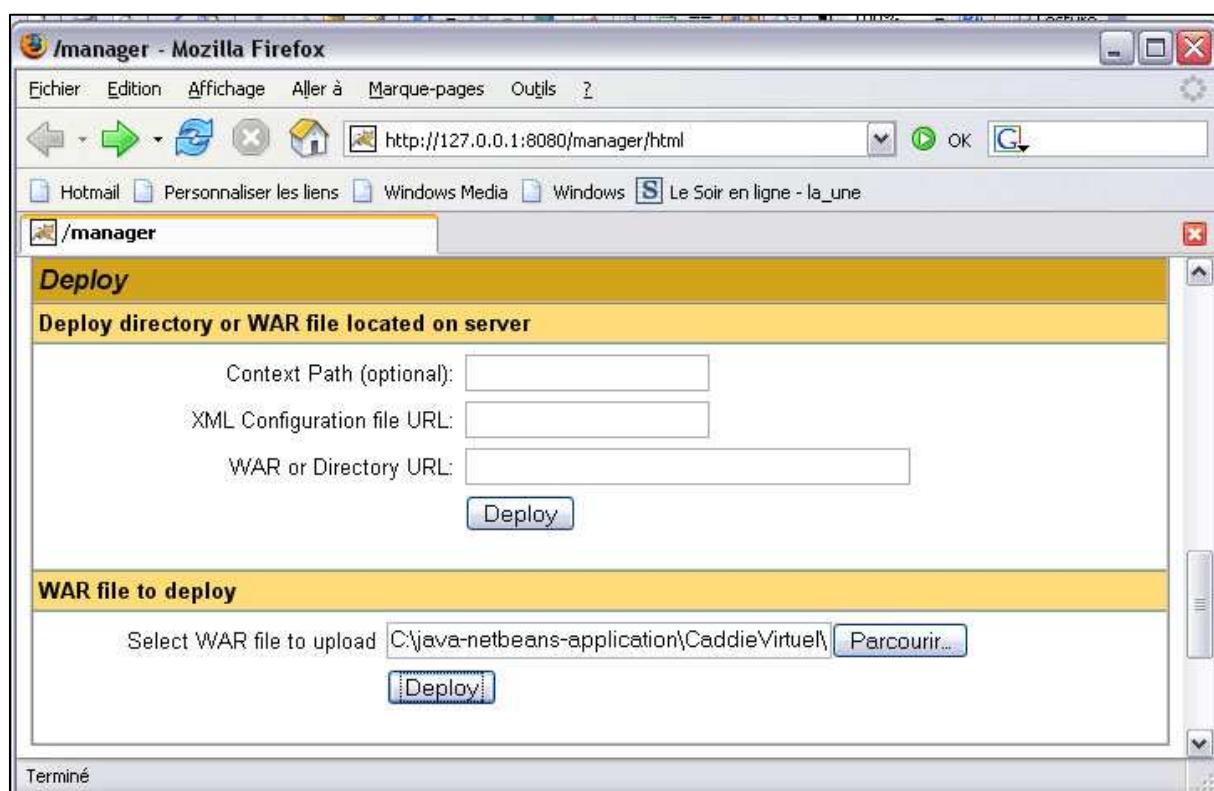
    <h1>JSP Page</h1>
    <a href="FormPage.html">Démarrer la servlet de base</a>
</body>
</html>
```

L'exécution du projet sous NetBeans donne :



7.6 Le déploiement avec Tomcat Manager

On peut invoquer depuis le menu Démarrer → Apache Tomcat 5.5 le Tomcat Manager. Il ne faut pas le confondre avec le module d'administration de Tomcat : ici, il est question de gérer les applications Web déployées sur le serveur (et l'une de ces applications est précisément celle d'administration). On trouve en bas de page un outil de déploiement d'applications Web. Dans le cas où l'on développé avec NetBeans, il n'y a qu'à choisir le WAR à utiliser :



Le bouton Deploy a pour effet de placer le fichier WAR dans C:\Program Files\Apache Software Foundation\Tomcat 6.0\webapps en utilisant le context.xml qui s'y trouve :

The screenshot shows the Mozilla Firefox browser window with the URL <http://127.0.0.1:8080/manager/html/start?path=/CaddieVirtuel>. The page displays the Apache Software Foundation logo and the Tomcat manager header 'Gestionnaire d'applications WEB Tomcat'. A message box at the bottom left says 'Message: OK – Application démarrée pour le chemin de contexte /CaddieVirtuel'. The main content area shows a table of deployed applications.

Applications					
Chemin	Nom d'affichage	Fonctionnant	Sessions	Commands	
/	Welcome to Tomcat	true	0	Démarrer	Arrêter Recharger Undeploy
/AchatWebStruts		true	0	Démarrer	Arrêter Recharger Undeploy
/CaddieVirtuel		true	0	Démarrer	Arrêter Recharger Undeploy
/admin	Tomcat Administration Application	true	0	Démarrer	Arrêter Recharger Undeploy
/host-manager	Tomcat Manager Application	true	0	Démarrer	Arrêter Recharger Undeploy
/manager	Tomcat Manager Application	true	0	Démarrer	Arrêter Recharger Undeploy
/tomcat-docs	Tomcat Documentation	true	0	Démarrer	Arrêter Recharger Undeploy

Un clic sur le lien "CaddieVirtuel" fait démarrer l'application, autrement dit lance le JSP index.jsp.

7.7 Une visualisation du trafic réseau

On peut se convaincre que la page dynamique reçue (et fabriquée par la servlet) occasionne un trafic réseau équivalent à celui du transfert d'une page Web statique. En effet, en utilisant un sniffer comme EthetDetect ou Ethereal :

#	Start Time	Client (IP:Port)	Server (IP:Port)	Protocol	packets	Last T
0	10:52:24.405	192.168.2.2:3015	192.168.2.1:8080	TCP	10	10:52

on retrouve⁶ les trois composantes *three-way handshake* de TCP (3 paquets), transaction HTTP (3 paquets : requête, réponse et ACK) et *four-way handshake* (4 paquets) :

Time Offset	Packet Len	Data Len	Data
10:52:24.405	62	0	
10:52:24.405	62	0	
10:52:24.405	54	0	
10:52:24.405	542	488	GET /servlet/FormServlet HTTP/1.1 Host: myria
10:52:24.405	427	373	HTTP/1.1 200 OK Content-Type: text/html;char
10:52:24.405	54	0	
10:52:44.702	60	0	
10:52:44.702	54	0	
10:52:45.718	54	0	
10:52:45.718	62	0	

Le premier paquet comportant des données représente bien la requête :

```
GET /servlet/FormServlet HTTP/1.1
Host: myriam:8080
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.0.1) Gecko/20020823 Netscape/7.0
Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,video/x-
mng,image/png,image/jpeg,image/gif;q=0.2,text/css,*/*;q=0.1
Accept-Language: en-us, en;q=0.50
Accept-Encoding: gzip, deflate, compress;q=0.9
Accept-Charset: ISO-8859-1, utf-8;q=0.66, *;q=0.66
Keep-Alive: 300
Connection: keep-alive
```

Le paquet suivant renvoie bien la réponse tout en acquittant la requête :

⁶ voir "Le protocole HTTP et le langage HTML" – du même auteur

```

HTTP/1.1 200 OK
Content-Type: text/html; charset=ISO-8859-1
Content-Length: 225
Date: Mon, 26 Jul 2004 09:02:07 GMT
Server: Apache-Coyote/1.1

<html>
<head>
<title>Servlet</title>
</head>
<body>
<HTML><HEAD><TITLE>
Réponse de la servlet à l'accès client
</TITLE></HEAD><BODY>
<H3>Woaaaawwww !</H3>
<b>Tu sais que tu es beau toi ;-) !!!</b>
</body>
</html>

```

Nous pouvons à présent nous concentrer sur la conception des servlets, et non plus sur leur déploiement. Commençons donc par bien prendre conscience d'une réalité

8. Un seul objet servlet pour plusieurs requêtes

Pour rappel, *une servlet est gérée par un moteur à servlets qui est multithread*. Il conviendra donc de penser aux problèmes des accès concurrents et de déclarer en synchronized les méthodes sensibles d'accès aux données partagées.

Pour illustrer ce propos, considérons une servlet presqu'aussi élémentaire que les précédentes, mais qui compte le nombre de requêtes dont elle fait l'objet. Elle devra bien sûr contenir un compteur comme variable membre et incrémenter ce compteur de manière atomique, puisque plusieurs threads peuvent invoquer la même servlet : cette incrémentation sera donc effectuée dans une section synchronized. Cette servlet est écrite à la dure sans les services du générateur de NetBeans :

CptServlet.java

```

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class CptServlet extends HttpServlet
{
    private int compteur = 0;

    public void doGet (HttpServletRequest request, HttpServletResponse response)
        throws IOException
    {
        PrintWriter out;
        response.setContentType("text/html");
        out = response.getWriter();
        out.println("<HTML><HEAD><TITLE>");
        out.println("Le compteur de requetes");
        out.println("</TITLE></HEAD><BODY>");

```

```
out.println("<H1>Réponse de la servlet</H1>");  
synchronized (this)  
{  
    compteur++;  
    out.println("Vous etes le client n° " + compteur + " !!!");  
}  
out.println("</BODY></HTML>");  
out.close();  
}  
}
```

On peut constater que des sollicitations différentes de la même servlet (browsers différents, rechargements) incrémentent toujours le même compteur :

The image displays three separate browser windows, each showing the output of a Java servlet named 'CptServlet'. The servlet's code is shown above, featuring a synchronized block that increments a counter ('compteur') and prints the client number. The three browser windows are:

- Mozilla Firefox:** Shows the message "Vous etes le client n° 1 !!!".
- Microsoft Internet Explorer:** Shows the message "Vous etes le client n° 2 !!!".
- Netscape:** Shows the message "Vous etes le client n° 4 !!!".

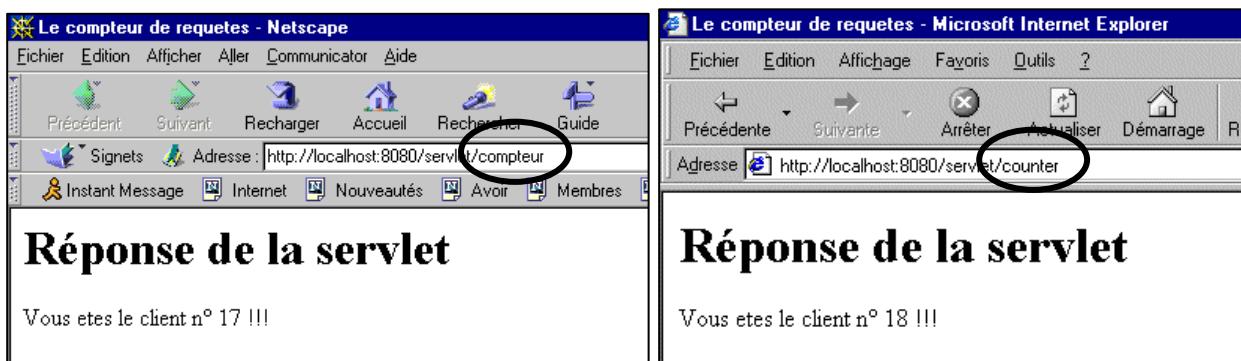
In all three cases, the browser title bar reads "Le compteur de requetes".

prouvant ainsi qu'**une seule instance de la classe CptServlet a bien été créée !**

Remarques

1) Il est possible d'imposer à une servlet d'être mono-thread en lui faisant implémenter l'interface **SingleThreadModel**. Cela ne coûte rien, puisqu'il n'y a aucune méthode à implémenter : cette implémentation permet simplement de signaler au serveur qu'il ne peut permettre l'exécution que d'une seule méthode service() à la fois. Ceci peut être utile dans le cas d'une servlet qui attaque une base de données. Les diverses requêtes seront néanmoins traitées simultanément, mais chaque requête provoquera l'instanciation d'un nouvel objet servlet, ... comme cela se passe pour les CGIs.

2) L'intérêt de donner à une servlet un nom différent de celui de la classe est que l'on peut en fait très bien donner plusieurs noms d'alias à une même classe, chaque alias pouvant ainsi se voir attribuer des paramètres d'initialisation différents :



3) Tomcat est bien multithread et rien n'empêche donc que le fichier web.xml parle de plusieurs servlets :

web.xml (avec deux servlets dont l'une à deux URLs)

```
<?xml version="1.0" encoding="UTF-8"?>

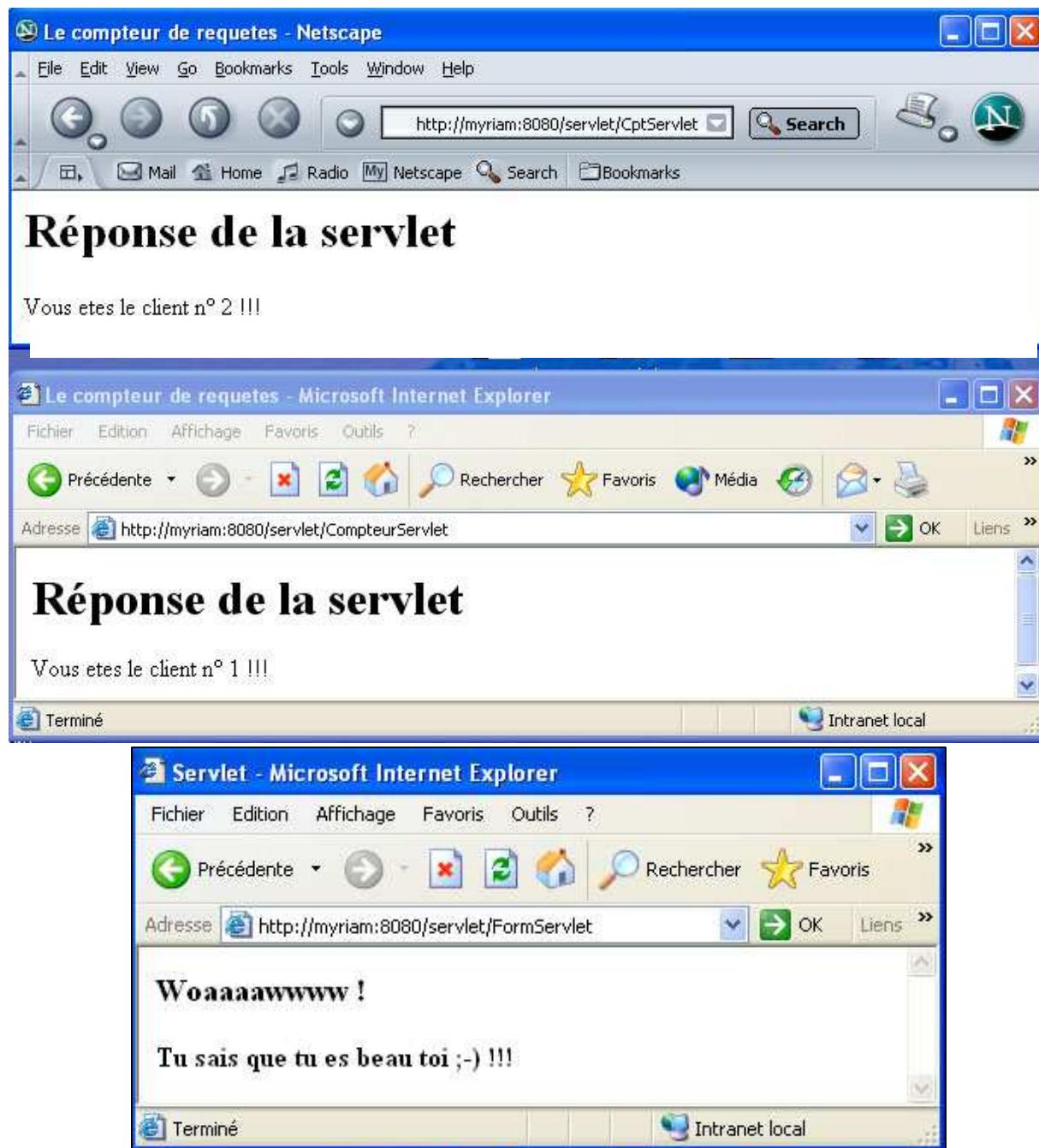
<!DOCTYPE web-app
PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
<servlet>
<servlet-name>FormServlet</servlet-name>
<servlet-class>FormServlet</servlet-class>
</servlet>
<servlet>
<servlet-name>CptServlet</servlet-name>
<servlet-class>CptServlet</servlet-class>
</servlet>

<servlet-mapping>
<servlet-name>FormServlet</servlet-name>
<url-pattern>/servlet/FormServlet</url-pattern>
</servlet-mapping>
```

```
<servlet-mapping>
<servlet-name>CptServlet</servlet-name>
<url-pattern>/servlet/CptServlet</url-pattern>
</servlet-mapping>
<servlet-mapping>
<servlet-name>CptServlet</servlet-name>
<url-pattern>/servlet/CompteurServlet</url-pattern>
</servlet-mapping>
...
</web-app>
```

On peut alors invoquer les servlets par les diverses URLs associées :



9. Une servlet d'informations

La servlet précédente ne fait guère usage du paramètre requête. Celui-ci permet cependant d'obtenir toute une série d'informations sur le client, et plus spécialement sur son browser, ainsi que sur le serveur lui-même. En effet, la classe HttpServletRequest implémente les méthodes de l'interface ServletRequest :

getAttribute(String)	getParameterNames()	getRemoteHost()
getCharacterEncoding()	getParameterValues(String)	getScheme()
getContentLength()	getProtocol()	getServerName()
getContentType()	getReader()	getServerPort()
getInputStream()	getRealPath(String)	
getParameter(String)	getRemoteAddr()	

augmentée de ses méthodes propres :

getAuthType()	getHeaderNames()	getQueryString()
getCookies()	getMethod()	getRemoteUser()
getDateHeader(String)	getPathInfo()	getRequestedSessionId()
getHeader(String)	getPathTranslated()	getServletPath()

Inutile de dire sans doute que ce sont *les headers du protocole HTTP* qui fournissent tous ces renseignements. La servlet suivante utilise quelques-unes de ces méthodes. Elle construit la page HTML de réponse en utilisant cette fois un flux ServletOutputStream :

TestInfoServlet.java

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

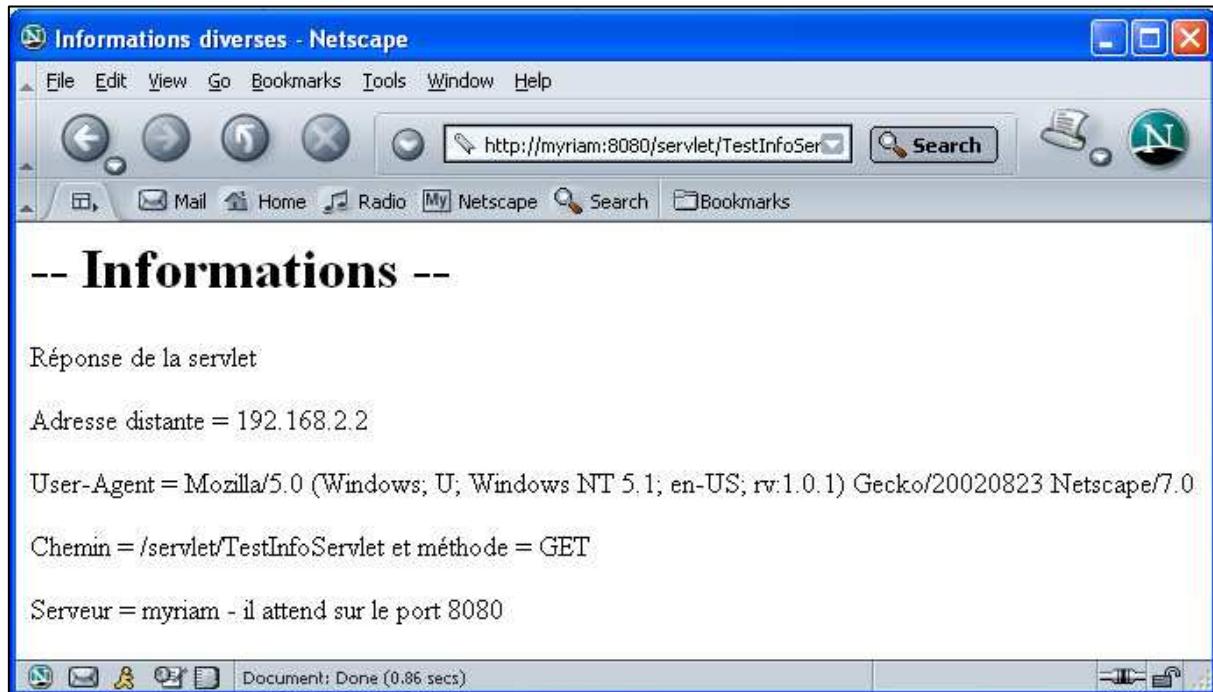
public class TestInfoServlet extends HttpServlet
{
    public void doGet (HttpServletRequest request, HttpServletResponse response)
        throws IOException
    {
        response.setContentType("text/html");
        ServletOutputStream out = response.getOutputStream();
        out.println("<HTML><HEAD><TITLE>");
        out.println("Informations diverses");
        out.println("</TITLE></HEAD><BODY>");
        String titre = "-- Informations --";
        out.println("<H1>" + titre + "</H1>");
        out.println("<p>Réponse de la servlet</p>");

        String remoteAddr = request.getRemoteAddr();
        out.println("<p>Adresse distante = " + remoteAddr + "</p> ");

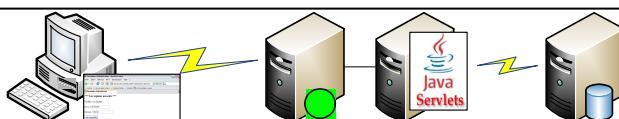
        String userAgent = request.getHeader("user-agent");
        if (userAgent != null) out.println("User-Agent = " + userAgent);
        else out.println("Le browser ne s'identifie pas");
        out.println("<p>");
    }
}
```

```
String methode = request.getMethod();
String path = request.getServletPath();
out.println("<p>Chemin = " + path + " et méthode = " + methode + "<p>");
String serveur = request.getServerName();
int port = request.getServerPort();
out.println("Serveur = " + serveur + " - il attend sur le port " + port + "<p>");
out.println("</BODY></HTML>"); out.close();
}
}
```

On obtient avec un browser et un serveur Tomcat sur une machine distante :



A. Requête avec un formulaire HTML
- Réponse de la servlet en page Web



10. Une servlet pour une requête paramétrée (méthode GET)

10.1 La récupération des données d'un formulaire

Nous allons à présent supposer que la requête envoyée au serveur, et visant en fait une servlet, comporte deux paramètres qui sont le nom et le prénom du client. L'utilisation typique est un **formulaire** qui demande tout d'abord l'entrée d'un nom et d'un prénom dans un formulaire d'une page HTML qui est ensuite envoyé au serveur.

On se souviendra (ou on découvrira) que les données d'un formulaire peuvent être récupérées par le serveur WEB

- ♦ selon la méthode **GET** : les données recueillies sont placées dans une variable d'environnement appelée typiquement **QUERY_STRING**. Evidemment, il y a intérêt à ce que le nombre de caractères qu'elle peut contenir soit suffisant, sans quoi une troncature des données sera effectuée.

- ♦ selon la méthode **POST** : ici, les données envoyées au serveur servent d'entrée au programme de traitement des données, exactement comme si elles étaient entrées au clavier – il s'agit donc d'une redirection des entrées.

Pour notre exemple, nous utiliserons la méthode par défaut GET. Le serveur saluera le client par son nom et lui donnera son numéro de client : autrement dit, la servlet comptera le nombre de connexions qui sont effectuées. Cela donne :

TrueFormServlet.java

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class TrueFormServlet extends HttpServlet
{
    int nbreConnexions;

    public void init (ServletConfig config) throws ServletException
    {
        super.init(config);
        nbreConnexions = 0;
    }

    public void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException
    {
        resp.setContentType("text/html");
        PrintWriter sortie = resp.getWriter();

        sortie.println("<HTML><HEAD><TITLE>");
        sortie.println("Réponse de la servlet à l'accès client");
        sortie.println("</TITLE></HEAD><BODY>");
        sortie.println("<H1>Accueil d'un nouveau client</H1>");
        sortie.println("<p>Bonjour cher " + req.getParameter("prenom") + " "
            + req.getParameter("nom") + " !!!<p>");

        synchronized (this) { nbreConnexions++; }

        sortie.println("<P><P>Vous etes notre client numéro " +
            Integer.toString(nbreConnexions) + " ...<p>");
        sortie.println("<p>Méthode utilisée = " + req.getMethod() + "<p>");
        sortie.println("</BODY></HTML>");
        sortie.close();
    }
}
```

Pour rappel, la méthode init() est exécutée la première fois que la servlet est chargée en mémoire. C'est donc l'endroit idéal pour initialiser des variables globales ou acquérir des

ressources indépendantes de la requête cliente; ici, il s'agit simplement d'initialiser le compteur de connexions.

L'objet qui implémente HttpServletRequest a pour tâche de fournir l'accès au header de la requête du client, ainsi qu'aux paramètres qui accompagnent cette requête. Pour cette raison, il possède une implémentation de la méthode :

```
public abstract String getParameter(String name)
```

qui fournit la valeur du paramètre correspondant au paramètre passé comme argument à la méthode (null si un tel paramètre n'existe pas).

10.2 Un test en local avec NetBeans

Nous pouvons plutôt créer notre servlet au sein de l'environnement NetBeans, comme indiqué plus haut. La servlet obtenue grâce au concours de l'EDI a l'aspect suivant :

TrueFormServlet.java (version NetBeans)

```
/*
 * FormServlet.java
 * Created on 24 mars 2004, 14:33
 */

package ServletsUtiles;

/**
 * @author Vilvens
 */

import java.io.*;
import java.net.*;

import javax.servlet.*;
import javax.servlet.http.*;

public class TrueFormServlet extends HttpServlet
{
    private int nbreConnexions = 0;

    /** Processes requests for both HTTP <code>GET</code> and <code>POST</code>
    methods.
     * @param request servlet request
     * @param response servlet response
     */
    protected void processRequest
        (HttpServletRequest request, HttpServletResponse response)
        throws ServletException, java.io.IOException
    {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
```

```
/* output your page here */
out.println("<html>");
out.println("<head>");
out.println("<title>Servlet typique : formulaire paramétré</title>");
out.println("</head>");
out.println("<body>");
out.println("<HTML><HEAD><TITLE>"); 
out.println("Réponse de la servlet à l'accès client");
out.println("</TITLE></HEAD><BODY>"); 
out.println("<H1>Accueil d'un nouveau client</H1>"); 
out.println("<p>Bonjour cher " + request.getParameter("prenom") + " "
+ request.getParameter("nom") + " !!!");

synchronized (this) { nbreConnexions++; }

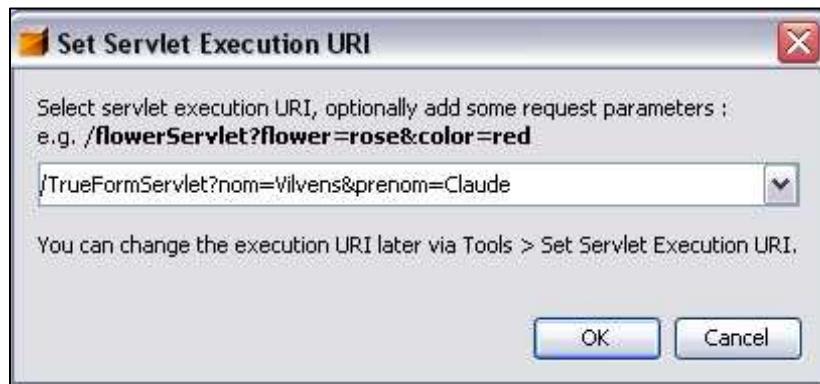
out.println("<P>Vous etes notre client numéro " +
    Integer.toString(nbreConnexions) + " ...");
out.println("<p>Méthode utilisée = " + request.getMethod() + "<p>");

out.println("</body>"); 
out.println("</html>"); 
out.close();
}

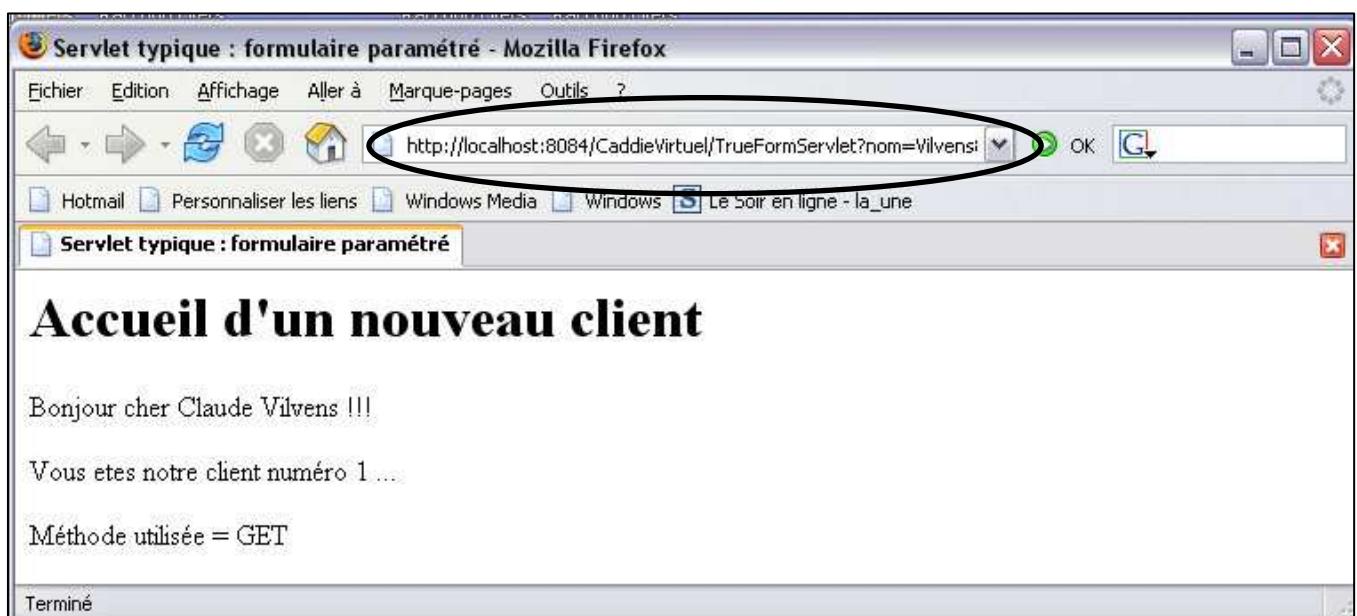
/** Handles the HTTP <code>GET</code> method.
 * @param request servlet request
 * @param response servlet response
 */
protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, java.io.IOException
{
    processRequest(request, response);
}
/** Handles the HTTP <code>POST</code> method.
 * @param request servlet request
 * @param response servlet response
 */
protected void doPost(HttpServletRequest request, HttpServletResponse response)
throws ServletException, java.io.IOException
{
    processRequest(request, response);
}
/** Returns a short description of the servlet.
 */
public String getServletInfo()
{
    return "Servlet exemple de traitement d'un formulaire";
}
```

Avant de faire exécuter notre servlet par Tomcat, on peut encore fixer la valeur des paramètres du formulaire en vue du test par la sélection dans le menu contextuel de l'objet servlet de :

Tools → Set Servlet Execution URI :



Si on oublie de le faire, l'exécution de la servlet provoquera l'apparition de la même boîte de dialogue. On obtiendra :



On remarquera les paramètres de la requête visibles dans l'URL. Bien sûr, on préférera certainement utiliser une page HTML (appelons-la PageFormulaire) créée dans l'environnement :

PageFormulaire.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">

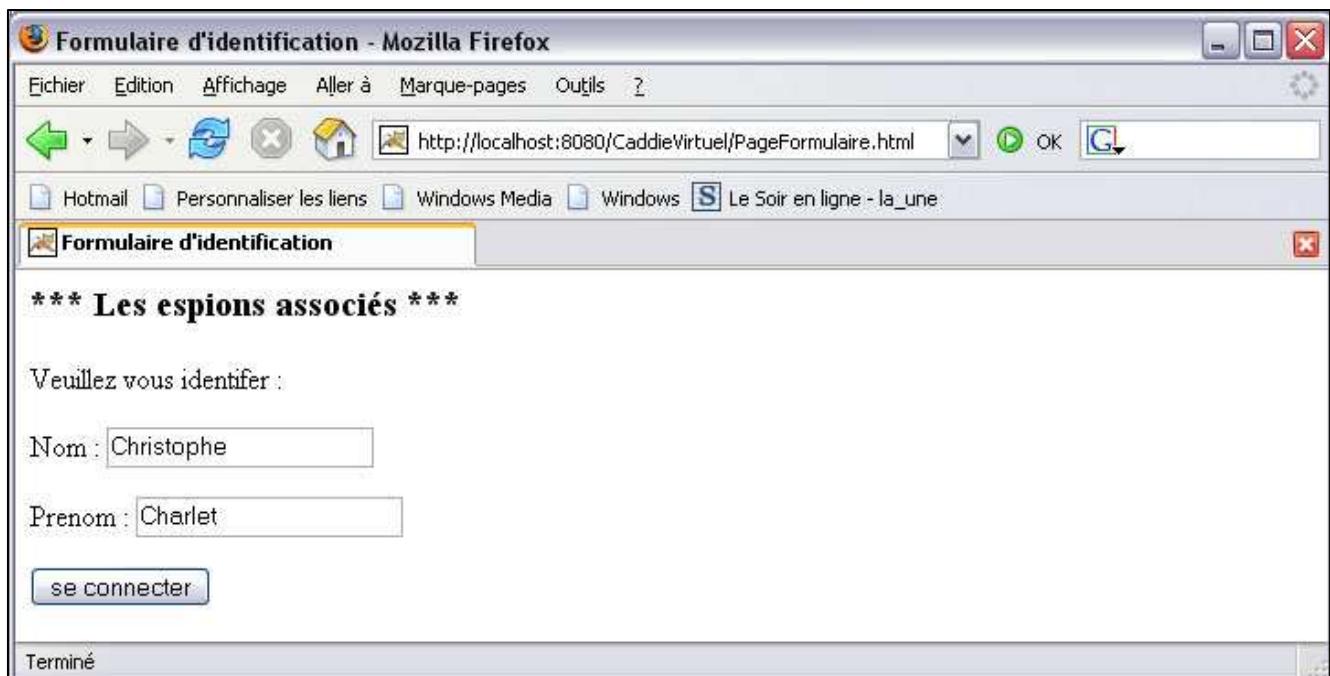
<HTML>
<HEAD>
<TITLE></TITLE>
</HEAD>
<BODY>
```

```
<H3>*** Les espions associ&eacute;s ***</H3>
<P>Veuillez vous identifier :
<BR>&nbsp;<BR>&nbsp;

<form method="GET" action="/CaddieVirtuel/TrueFormServlet">
<P>Nom : <input type="text" name="nom" size=20></P>
<P>Prenom : <input type="text" name="prenom" size=20></P>
<P><input type="submit" value="se connecter"></P>
</form>

</BODY>
</HTML>
```

Le choix d'Execute dans le menu contextuel associé à la page donne :

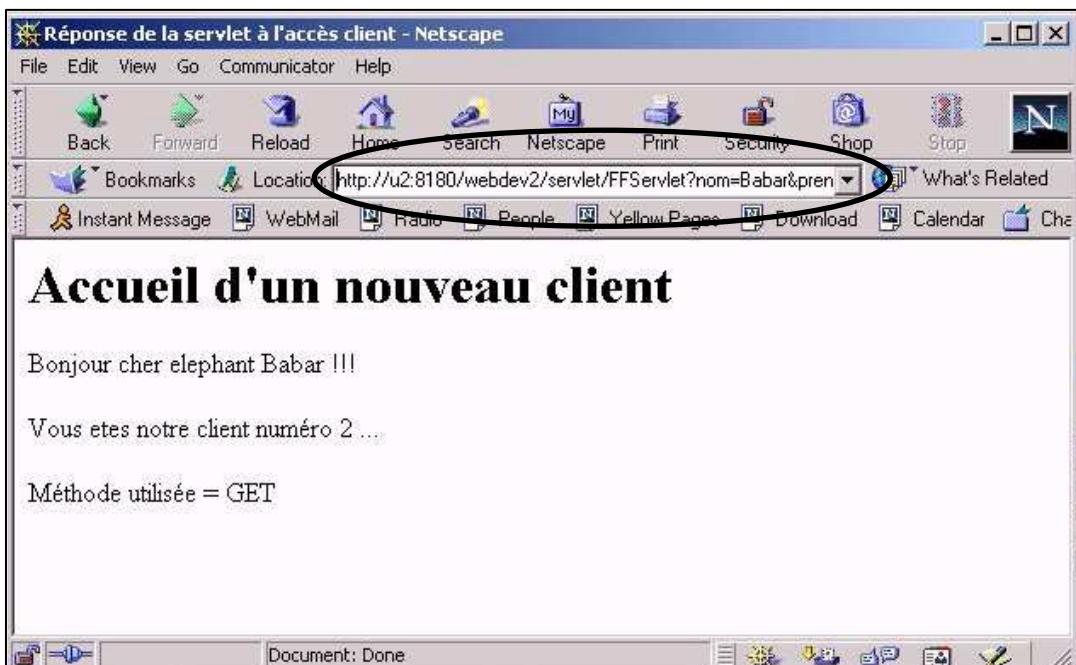


10.3 Un test à distance sur une machine Linux

Nous allons à présent utiliser un Tomcat sur une machine serveur Linux (ici, U2). Ce tomcat est configuré selon un server.xml tel qu'exposé au paragraphe 5.4, avec un contexte par utilisateur. Dans le cas de l'utilisateur Vilvens, il utilisera donc dans ses URLs l'alias "/webdev2". Le répertoire WEB-INF sera créé dans le répertoire associé à cet alias webdev2, soit ici /home/vilvens/webdev. Il suffira alors

- ◆ de placer la servlet dans le répertoire prévu par défaut, soit le répertoire classes du répertoire WEB-INF,
- ◆ de mettre à jour le fichier web.xml;

pour obtenir le même résultat :



1 Une servlet pour une requête paramétrée (méthode POST)

Le même formulaire peut être utilisé, mais avec la méthode POST. La méthode

```
protected void doPost(HttpServletRequest req, HttpServletResponse resp)  
throws ServletException, IOException
```

permet de réaliser les opérations de type POST pour le traitement d'un formulaire.

Il suffit de substituer cette méthode à doGet() dans notre servlet – dans le cas d'une servlet générée par NetBeans, il n'y a rien à faire puisque tout passe par la méthode processRequest(). On obtient alors :



Cette fois, les paramètres n'apparaissent plus en clair dans la zone d'URL du browser. De plus, à la différence de la méthode GET, la méthode POST ne limite pas le nombre de caractères occupés par ceux-ci.

11. Une servlet avec JDBC

Nous en arrivons à l'utilisation classique d'une servlet : elle est contactée par un client WEB afin de réaliser une recherche dans une base de données. Celle-ci sera accédée par un pont JDBC. Fixons les idées en utilisant une base connue de type MS-Access sous ODBC par le nom de source (DSN) "helvetes". Pour rester dans le simple, nous allons reprendre la page HTML évoquée au point précédent, si ce n'est que la servlet contactée est la suivante (version "from scratch", non générée) :

JdbcServlet.java

```

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.sql.*;

public class JdbcServlet extends HttpServlet
{
    public void doGet(HttpServletRequest req,
                      HttpServletResponse resp) throws ServletException, IOException
    {
        resp.setContentType("text/html");
        PrintWriter sortie = resp.getWriter();

        sortie.println("<HTML><HEAD><TITLE>");
        sortie.println("Réponse de la servlet à l'accès client");
        sortie.println("</TITLE></HEAD><BODY>");
        sortie.println("<H1>Accueil d'un nouveau client</H1>");
        sortie.println("<p>Bonjour très cher " + req.getParameter("prenom") + " "
                     + req.getParameter("nom") + " !!!<p>");
        sortie.println("<p>Méthode utilisée = " + req.getMethod() + "<p>");

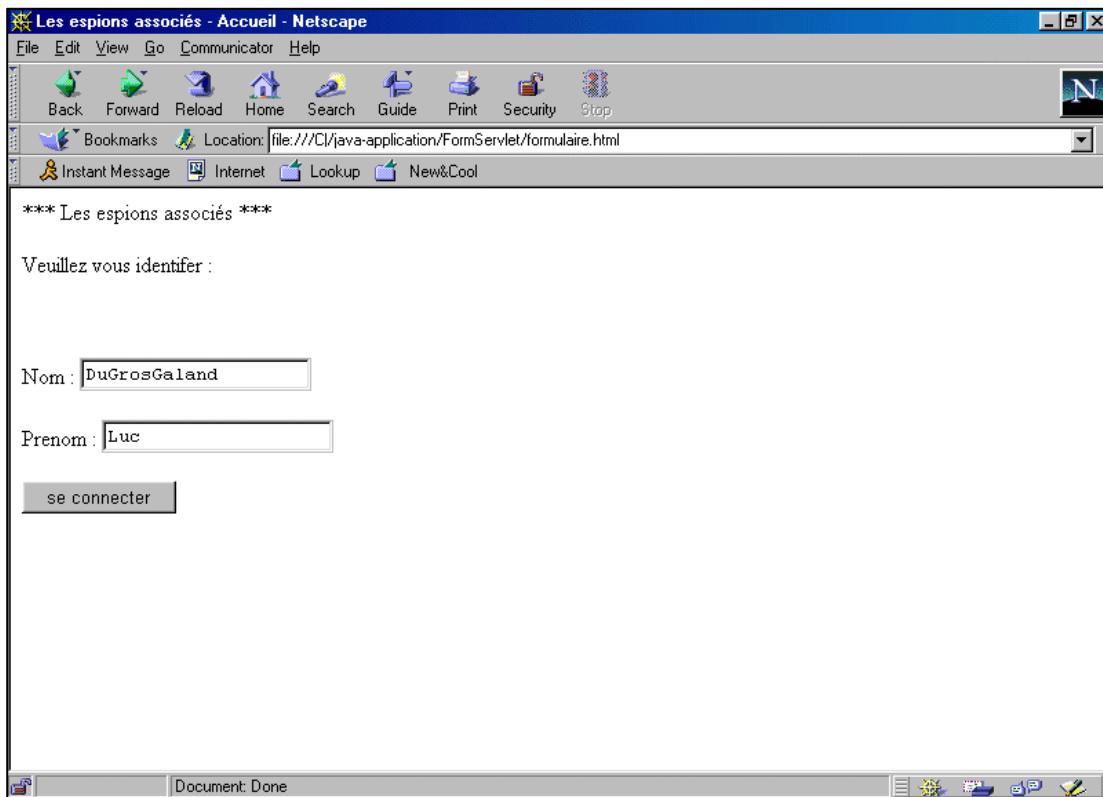
        Connection con;
        Statement instruc;
        ResultSet rs;

        try { Class.forName("sun.jdbc.odbc.JdbcOdbcDriver"); }
        catch (ClassNotFoundException e) { sortie.println("Driver JDBC-OBDC non chargé"); }
        try
        {
            con = DriverManager.getConnection("jdbc:odbc:helvetes","","");
            sortie.println("<P>Connexion à la BDD helvetes réalisée</P>");
            instruc = con.createStatement();
            rs = instruc.executeQuery("select * from clients");
            sortie.println("<p>Après le query<p>");
            int cpt = 0;
            while (rs.next())
            {
                cpt++;
                String nC = rs.getString("numClient"); String n = rs.getString("nom");
                String p = rs.getString("prenom"); int s = rs.getInt("solde");
                sortie.println("<P>" + cpt + ". " + nC + " : " + n + " " + p + "/" + s + "-- </P>");
            }
        }
        catch (SQLException e)
        {
            sortie.println("<P>Erreur JDBC-OBDC : " + e.getMessage() + " ** " +
                          e.getSQLState() + "</P>");
        }
        sortie.println("</BODY></HTML>"); sortie.close();
    }
}

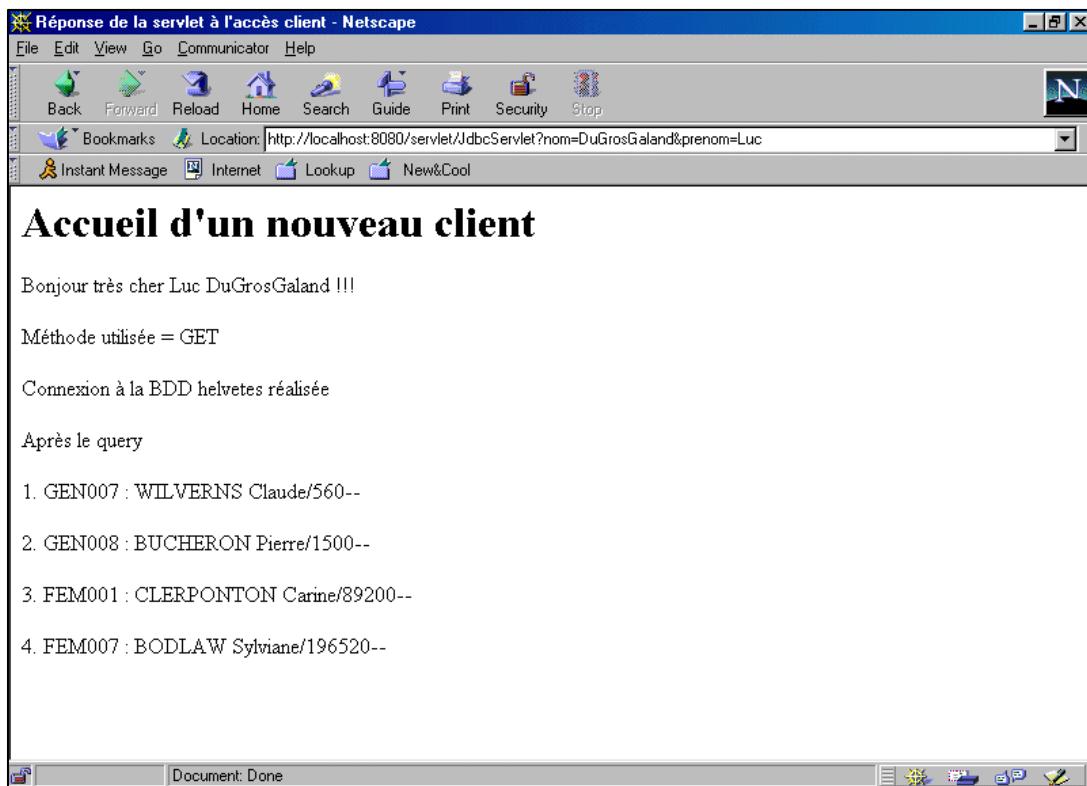
```

```
}
```

Si donc la page HTML est la suivante :



le résultat de "se connecter" sera :



12. Les paramètres d'initialisation d'une servlet

12.1 L'initialisation d'une servlet

Il est toujours possible de spécifier des paramètres complémentaires qui représentent des valeurs initiales pour une servlet donnée. Celle-ci en prendra connaissance à son démarrage, donc typiquement, au sein de sa méthode **init()**, au moyen de la méthode :

```
public String getInitParameter(String name)
```

le paramètre permettant de désigner le paramètre initial dont on désire récupérer la valeur; celle-ci sera donc toujours récupérée sous forme de String. En fait, cette méthode appartient à l'interface **ServletConfig** (package javax.servlet), et la classe **GenericServlet**, dont **HttpServlet** hérite, implémente cet interface.

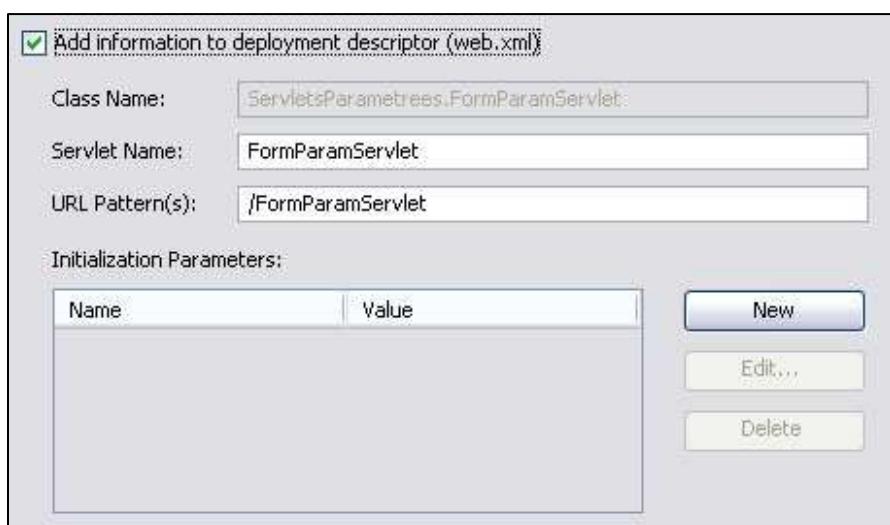
Nous allons écrire une servlet **FormParamServlet** (package : ServletsParametrees et nom utilisé : AccesClients) qui ressemble assez à l'une des précédentes, si ce n'est qu'*elle va lire dans ses paramètres initiaux le nom, le mot de passe et le nom d'une source de données pour JDBC*.

1.1 Les paramètres d'initialisation pour Tomcat

Dans le cas de Tomcat, les paramètres d'initialisation se spécifient évidemment dans le fichier web.xml. On utilise pour cela, au sein de la zone <servlet></servlet>, une ou plusieurs fois les tags :

```
<init-param>
    <param-name>nom_du_paramètre</param-name>
    <param-value>valeur_du_paramètre </param-value>
</init-param>
```

Donc, pour notre exemple développé sous NetBeans, nous préciserons ces paramètres dès la création de la servlet :



Les appuis successifs sur les boutons New et Edit permettent de créer les paramètres d'initialisation :



pour obtenir finalement :

Initialization Parameters:	
Name	Value
nomAutorise	vil
motDePasse	pataclix
sourceBase	helvetes

et

web.xml (avec paramètres d'initialisation)

```
<?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">
  <servlet>
    <servlet-name>FormParamServlet</servlet-name>
    <servlet-class>ServletsParametrees.FormParamServlet</servlet-class>
    <init-param>
      <param-name>nomAutorise</param-name>
      <param-value>vil</param-value>
    </init-param>
    <init-param>
      <param-name>motDePasse</param-name>
      <param-value>pataclix</param-value>
    </init-param>
    <init-param>
      <param-name>sourceBase</param-name>
      <param-value>helvetes</param-value>
    </init-param>
  </servlet>
  <servlet-mapping>
    <servlet-name>FormParamServlet</servlet-name>
    <url-pattern>/AccesClients</url-pattern>
  </servlet-mapping>
  <session-config><session-timeout>30</session-timeout></session-config>
  <welcome-file-list> <welcome-file>index.jsp </welcome-file></welcome-file-list>
</web-app>
```

On remarquera que l'URL d'appel (clause <url-pattern>) ne présente plus la forme habituelle : ce sera cette fois "AccesClients". Ce sera donc cette URL qui sera utilisée dans la page HTML contenant le formulaire :

FormulaireAccessHelvetes.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head><title></title></head>
  <body>
    <H3>Accès à la base de données des clients </H3>
    <form method="post" action="AccesClients">
      <p>Nom :<br/><input type="text" name="nom" size="20"> </p>
      <p>Mot de passe :<br/><input type="text" name="pwd" size="20"> </p>
      <p><input type="submit" value="se connecter"> <br><br></p>
    </form>
  </body>
</html>
```

12.2 Une servlet avec paramètres d'initialisation

Notre servlet va donc lire les valeurs initiales, dont le nom et le mot de passe de l'utilisateur autorisé à accéder à la base de données (le gestionnaire). Elle comparera ces informations avec celles fournies par un utilisateur qui tente de se connecter sur le serveur : en cas d'égalité, la base sera accédée (dans le cas d'ODBC, la source de données doit évidemment être une *source de données système*, puisque nous allons travailler en réseau); sinon, l'accès sera refusé. Générée avec NetBeans, cela donne :

FormParamServlet.java

```
package ServletsParametrees;

import java.io.*;
import java.net.*;
import java.sql.*;
import javax.servlet.*;
import javax.servlet.http.*;

/**
 * @author Vilvens
 */

public class FormParamServlet extends HttpServlet
{
    private String gestionnaire;
    private String pwdGestionnaire;
    private String sourceDonnees;

    public void init (ServletConfig c) throws ServletException
    {
        super.init(c);
    }
}
```

```

gestionnaire = getInitParameter("nomAutorise");
System.out.println(gestionnaire);
pwdGestionnaire = getInitParameter("motDePasse");
sourceDonnees = getInitParameter("sourceBase");
}

protected void processRequest(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException
{
    response.setContentType("text/html;charset=UTF-8");
    PrintWriter out = response.getWriter();
    out.println("<html>"); out.println("<head>");
    out.println("<title>Réponse de la servlet à l'accès client</title>");
    out.println("</head>"); out.println("<body>");
    out.println("<H1>Accueil d'un nouveau client</H1>");
    String user = request.getParameter("nom");
    out.println("<p>Bonjour très cher " + user + " !!!<p>");
    String pwdUser = request.getParameter("pwd");
    out.println("<p>Méthode utilisée = " + request.getMethod() + "<p>");
    if (!user.equals(gestionnaire) || !pwdUser.equals(pwdGestionnaire))
    {
        out.println("<H3>Vous n'êtes pas le gestionnaire - désolé</H3>");
        out.println("</BODY></HTML>");
        out.close();
    }
    out.println("<H3>Vous êtes bien gestionnaire - bonjour vous !</H3>");
    out.println("Gestionnaire = " + gestionnaire + "<p>");
    out.println("pwdGestionnaire = " + pwdGestionnaire + "<p>");
    out.println("sourceDonnees = " + sourceDonnees + "<p>");

    Connection con;
    Statement instruc;
    ResultSet rs;
    try
    {
        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    }
    catch (ClassNotFoundException e)
    {
        out.println("Driver JDBC-OBDC non chargé");
    }
    try
    {
        StringBuffer source = new StringBuffer("jdbc:odbc:");
        source.append(sourceDonnees);
        String sourceS = new String(source);
        con = DriverManager.getConnection(sourceS,"","");
        out.println("<P>Connexion à la BDD helvetes réalisée</P>");
        instruc = con.createStatement();
        rs = instruc.executeQuery("select * from clients");
    }
}

```

```

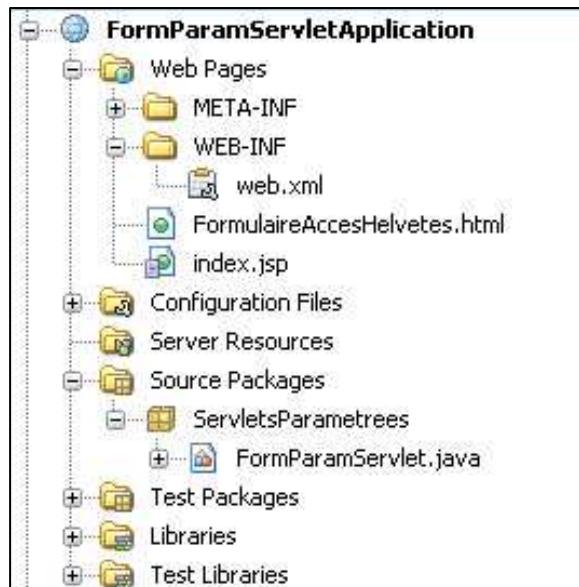
out.println("<p>Après le query</p>");
int cpt = 0;

while (rs.next())
{
    cpt++;
    String nC = rs.getString("numClient");
    String n = rs.getString("nom");
    String p = rs.getString("prenom");
    int s = rs.getInt("solde");
    out.println("<P>" + cpt + ". " + nC + " : " + n + " " + p + "/" + s + "-- </P>");
}
catch (SQLException e)
{
    out.println("<P>Erreur JDBC-OBDC</P>");
}
out.println("</body>");
out.println("</html>");
out.close();
}

protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException
{ processRequest(request, response); }
protected void doPost(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException
{ processRequest(request, response); }
public String getServletInfo()
{
    return "Servlet avec paramètres d'initialisation";
}
}

```

Le projet au sein de NetBeans comporte alors:



L'exécution de la page (si on peut dire cela) donne :

Nom : vil

Mot de passe : genius

se connecter

Terminé

Avec un utilisateur quelconque :

tandis qu'avec le vrai gestionnaire:

Réponse de la servlet à l'accès client - Mozilla Firefox

Accueil d'un nouveau client

Bonjour très cher vil !!!

Méthode utilisée = POST

Vous n'etes pas le gestionnaire - désolé

Terminé

Réponse de la servlet à l'accès client - Mozilla Firefox

Accueil d'un nouveau client

Bonjour très cher vil !!!

Méthode utilisée = POST

Vous etes bien gestionnaire - bonjour vous !

Terminé

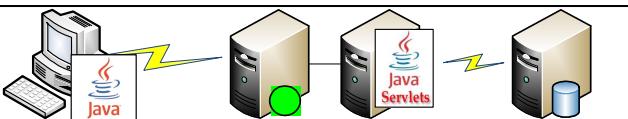
Il en est de même avec un serveur Tomcat installé sur une machine distante : il suffit de placer le war généré par NetBeans dans le répertoire webapps. Un emachine distante peut alors invoquer la page de démarrage :

The top screenshot shows a Firefox window with the URL <http://192.168.1.4:8080/FormParamServletApplication/FormulaireAccesHelvetes.html>. The page title is "Accès à la base de données des clients". It contains a form with fields "Nom" (containing "vil") and "Mot de passe" (containing "pataclac"), and a "se connecter" button. The status bar at the bottom says "Done".

The bottom screenshot shows a Firefox window with the URL <http://192.168.1.4:8080/FormParamServletApplication/AccesClients>. The page title is "Réponse de la servlet à l'accès client - Mozilla Firefox". It displays the message "Accueil d'un nouveau client", "Bonjour très cher vil !!!", "Méthode utilisée = POST", and "Vous n'êtes pas le gestionnaire - désolé". The status bar at the bottom says "Done".

Et si nous placions un peu d'intelligence dans nos pages Web ? C'est bien cela, des applets ...

B. Requête avec un formulaire en applet
- Réponse de la servlet en page Web



13. La communication applet-servlet

Selon le schéma classique, l'applet représente la partie active d'un "thin" client tandis que la servlet représente la partie intelligente du côté du serveur.

13.1 La communication par sockets

La communication, à la base, s'effectue en utilisant TCP/IP et l'on pourrait penser que le seul paradigme de communication applet-servlet consiste à utiliser les sockets TCP/IP, avec toute la souplesse que l'on peut souhaiter. Cependant, il faut remarquer que :

- ◆ ce procédé ne peut fonctionner pour des applets issues d'un serveur placé derrière un firewall; en effet, la plupart des firewalls n'autorisent pas la connexion par socket; la solution semble donc mieux convenir à un Intranet;
- ◆ il faudra probablement définir un protocole applicatif propre afin de régler la communication;
- ◆ les browsers sont construits sur HTTP; l'utilisation d'un autre protocole complique donc les choses.

13.2 La communication par HTTP

A l'inverse, une communication soutenue directement par HTTP

- ◆ fonctionne pour toutes les applets, que leur serveur soit derrière un firewall ou pas; on parle encore de "tunnel HTTP" dans le cas où il s'agit en fait de requêtes qui ne sont pas des requêtes HTTP, mais qui sont encapsulées dans une trame HTTP, utilisant ainsi les caractéristiques préétablies de celui-ci (évidemment, c'est un peu plus lent);
- ◆ permet à une applet de communiquer avec un programme de type CGI écrit en n'importe quel langage et pas seulement en Java (c'est un peu vrai aussi pour la communication pas sockets, mais au prix de grosses difficultés);
- ◆ la partie serveur, donc le CGI, peut aussi bien fonctionner pour une simple page Web que pour une applet.

Evidemment, rien n'est parfait et l'on peut pointer comme inconvénients de la communication HTTP les points suivants :

- ◆ le procédé est lent puisque, HTTP étant un protocole sans état, il faut établir un nouveau canal de communication à chaque requête/réponse;
- ◆ les requêtes doivent être construites selon les couples <nom donnée>=<valeur donnée>;
- ◆ seule l'applet peut débuter la communication.

14. Un formulaire dans une applet : la méthode GET

14.1 L'utilisation d'une applet en local

Nous allons à présent mettre sur pied une première étape vers un dialogue applet-servlet. Ici, nous allons remplacer le formulaire HTML utilisé pour tester la servlet FormParamServlet par une applet qui présente un interface similaire. Que gagne-t-on à ce changement ? Certainement de la sécurité puisque qu'un texte statique (les tags HTML) à la merci du premier hacker venu est remplacé par du code Java, avec tout ce que cela implique comme contrôles de sécurité !

Notre applet va donc proposer un interface du type suivant :

The screenshot shows a Java applet window with a light gray background. At the top center, it says "Accueil : Bienvenue :)" in red font. Below that, there are two text input fields. The first field has the placeholder "Votre nom:" and contains the text "vil". The second field has the placeholder "Votre mot de passe:" and contains the text "pataclix". At the bottom left, there is a small label "A vous:". To its right is a blue rectangular button with the white text "Se connecter".

Ce n'est évidemment pas ceci qui constitue la nouveauté ! Ce qui est neuf, c'est la manière dont nous allons activer la servlet FormParamServlet et lui transmettre les données entrées dans les zones de texte. Lorsque le bouton "Se connecter" est enfoncé, l'exécution passe dans la méthode actionPerformed() que nous aurons pris soin de créer (ou plutôt dans son substitut

créé par NetBeans – mais c'est un détail). Notre problème est d'y programmer l'équivalent de l'appel :

"<http://localhost:8080/FormParamServletApplication/AccesClients?nom=Bond&pwd=James>"

si on a entré "Bond" et "James" dans les zones de texte. Il est clair qu'il nous faut tout d'abord créer un objet URL contenant l'appel, ce que nous ferons en lui passant la chaîne de caractères correspondante. Mais comment construire cette chaîne ? Bien sûr, nous allons récupérer les nom et prénom dans les zones de texte; si nom et prenom sont ces dernières :

```
String n = nom.getText();
String p = pwd.getText();
```

Pour fabriquer la chaîne de caractères souhaitée, il faut bien sûr insérer les "?" et "&", mais aussi tenir compte du fait que, dans la chaîne envoyée par le client :

- ◆ un blanc est remplacé par '+';
- ◆ les caractères spéciaux, comme '\$' ou les caractères accentués, sont automatiquement remplacés par le browser en leur forme hexadécimale, soit "%XX".

En fait, la chaîne initiale doit être convertie en un code **MIME** appelé, assez logiquement, "**x-www-form-urlencoded**". Fort heureusement, il existe une classe **URLEncoder**, du package java.net, qui se charge du travail au moyen de ses méthodes :

- ◆ public static String **encode** (String s)
- ◆ public static String **encode** (String s, String enc) throws UnsupportedEncodingException

la première, plus simple, étant cependant deprecated depuis le JDK 1.4. Nous pourrons donc construire la chaîne, qui permettra de créer l'URL, en écrivant :

```
String adresseServlet = "http://localhost:8080/FormParamServletApplication/AccesClients"
    + "?nom=" + URLEncoder.encode(n)
    + "&prenom=" + URLEncoder.encode(p);
```

L'URL elle-même se construira alors sans problème :

```
URL urlServlet = null;
try
{
    urlServlet = new URL (adresseServlet);
}
catch (MalformedURLException ex)
{
    System.out.println("Aie aie - URL louche ! : " + ex.getMessage());
}
```

Muni de cette URL, nous pouvons à présent appeler la servlet comme n'importe quelle page HTML statique. La méthode

```
public abstract void showDocument(URL url)
```

de l'interface AppletContext permet de réaliser cet appel : pour rappel, elle remplace la page courante par la page dont l'URL est spécifiée comme paramètre. L'interface **AppletContext** caractérise tout ce qui fait l'environnement d'utilisation de l'applet, soit pour nous la page HTML et son support qu'est le browser. Une implémentation nous en sera fournie par la méthode de la classe Applet :

```
public AppletContext getAppletContext()
```

qui fournit l'environnement effectif. Notre applet (avec du code généré par NetBeans) s'écrit donc en définitive :

FormApplet.java

```
/*
 * FormApplet.java
 */

package AppletsCommunicateServlets;

import java.net.*;

/**
 * @author Vilvens
 */

public class FormApplet extends javax.swing.JApplet
{
    public void init()
    {
        try
        {   java.awt.EventQueue.invokeLater(new Runnable()
            {   public void run() { initComponents(); } });
        }
        catch (Exception ex)
        { ex.printStackTrace(); }
    }

    private void initComponents() { ... }

    private void BConnecterPerformed(java.awt.event.ActionEvent evt)
    {
        String n = nom.getText();
        String p = pwd.getText();
        String adresseServlet =
            "http://localhost:8080/FormParamServletApplication/AccesClients"
            + "?nom=" + URLEncoder.encode(n)
            + "&pwd=" + URLEncoder.encode(p);
        System.out.println("Url de la servlet = "+adresseServlet);
    }
}
```

```

URL urlServlet = null;
try
{
    urlServlet = new URL (adresseServlet);
}
catch (MalformedURLException ex)
{
    System.out.println("Aie aie - URL louche ! : " + ex.getMessage());
}
getAppletContext().showDocument(urlServlet);
}

private javax.swing.JButton BConnecter;
private javax.swing.JLabel jLabel1;
private javax.swing.JLabel jLabel2;
private javax.swing.JLabel jLabel3;
private javax.swing.JLabel jLabel4;
private javax.swing.JTextField nom;
private javax.swing.JTextField pwd;
}

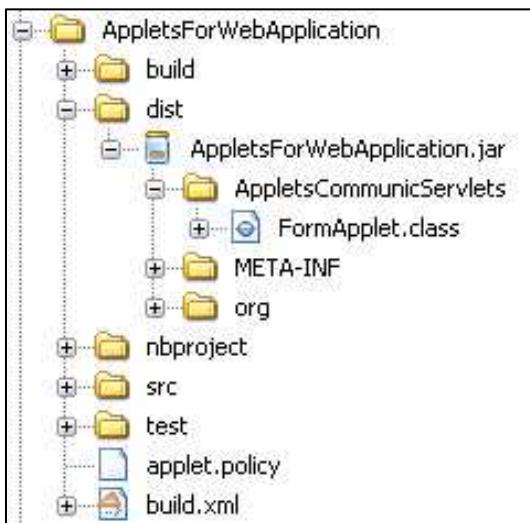
```

tandis que la servlet reste inchangée.

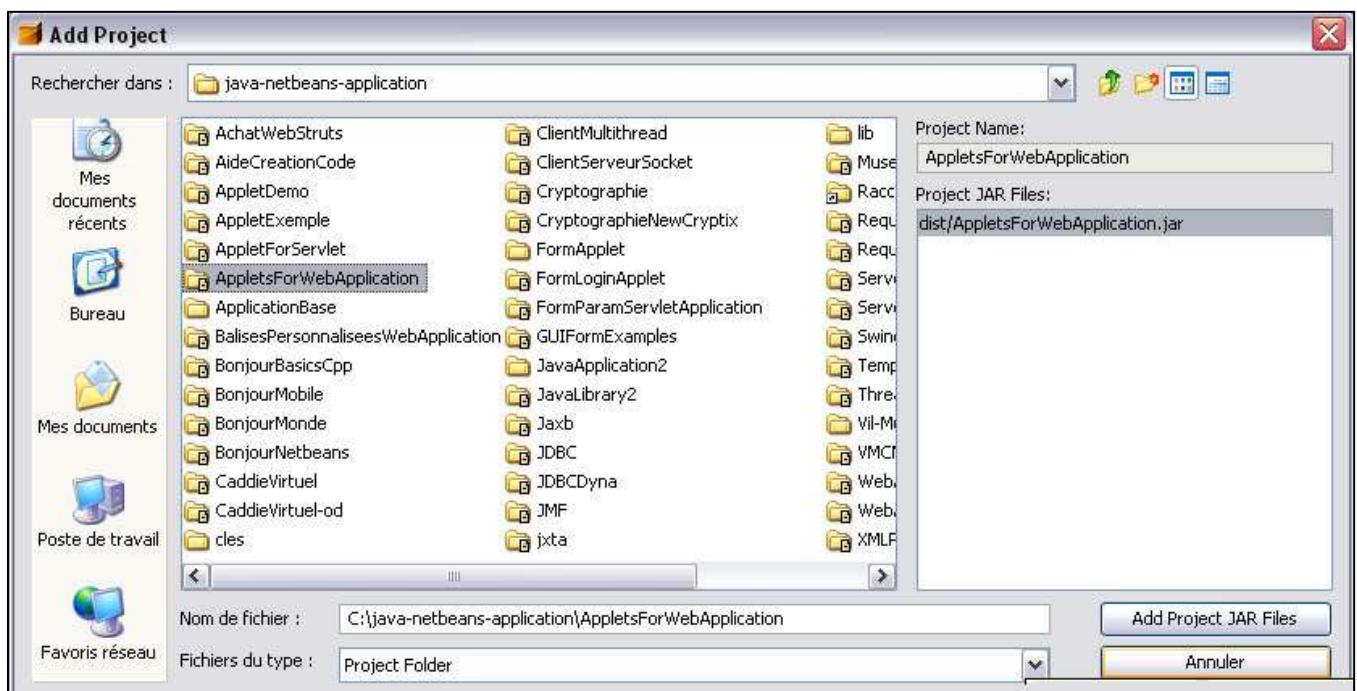
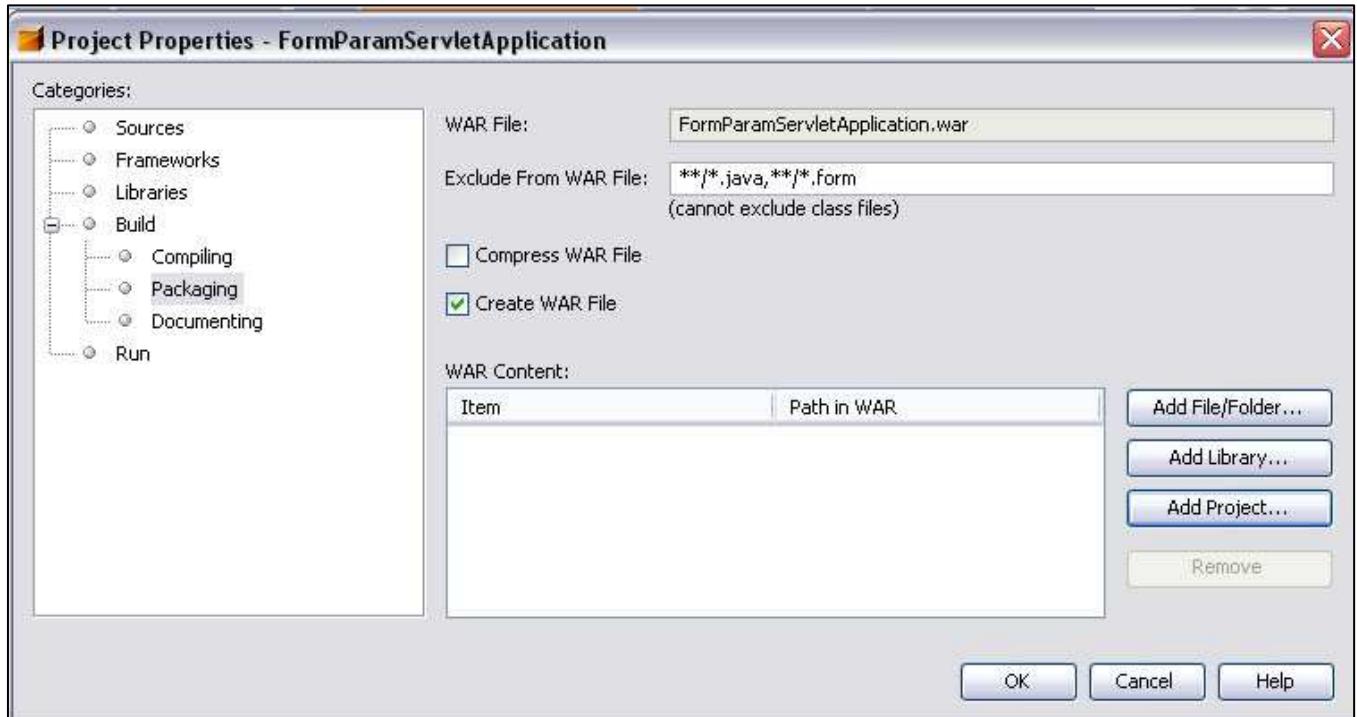
14.2 Le déploiement de l'applet dans l'application Web

En pratique, l'applet et la page HTML qui va la contenir doivent logiquement faire partie de l'application Web, donc du fichier war, en compagnie de la servlet. Cela ne pose aucun problème pour la page HTML. Par contre, il faut savoir que *l'applet ne pourra faire partie de l'application Web que si elle y est intégrée sous forme de jar*. Une fois notre applet finalisée dans un projet Netbeans différent (disons AppletsForWebApplication), il nous faut donc :

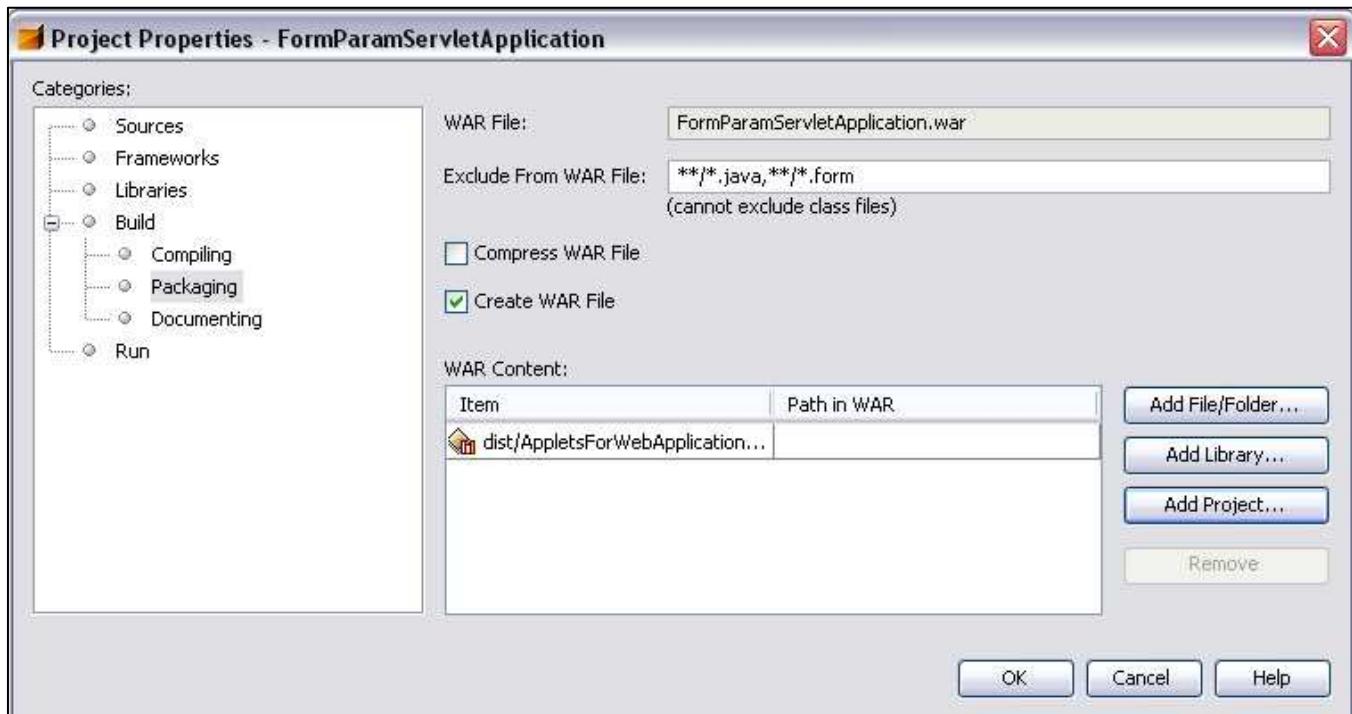
- ♦ générer le jar contenant l'applet : il suffit pour cela de réaliser un Build du projet AppletsForWebApplication, le jar résultant AppletsForWebApplication.jar se trouvant dans le répertoire dist habituel; on en arrive à ceci :



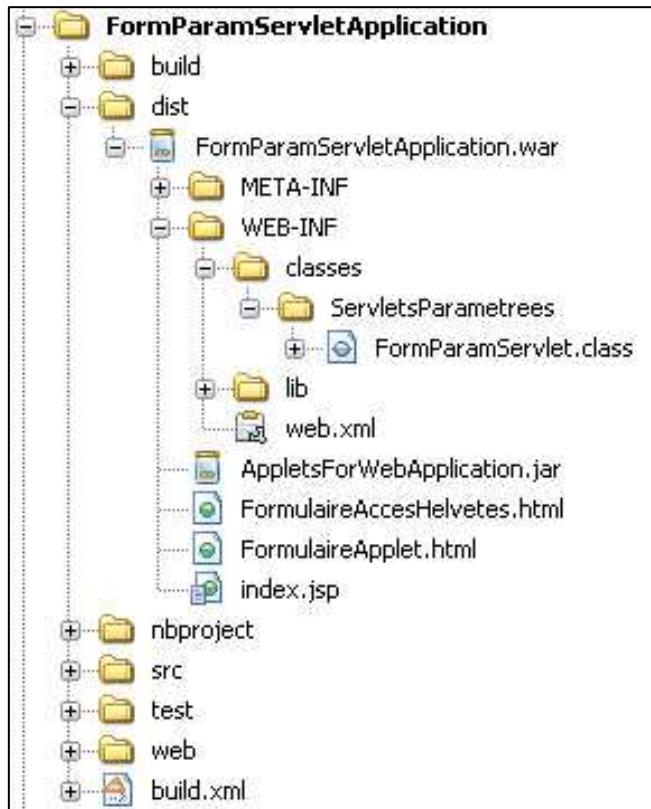
- ♦ intégrer ce jar dans le war de l'application Web FormParamServletApplication; pour cela, on choisit Properties dans le menu contextuel obtenu par un clic droit sur le projet FormParamServletApplication – c'est le noeud Build-Packaging qui nous intéresse :



L'appui sur le bouton Add Project JAR Files donne :



Il reste à conclure par OK pour en arriver à ceci :



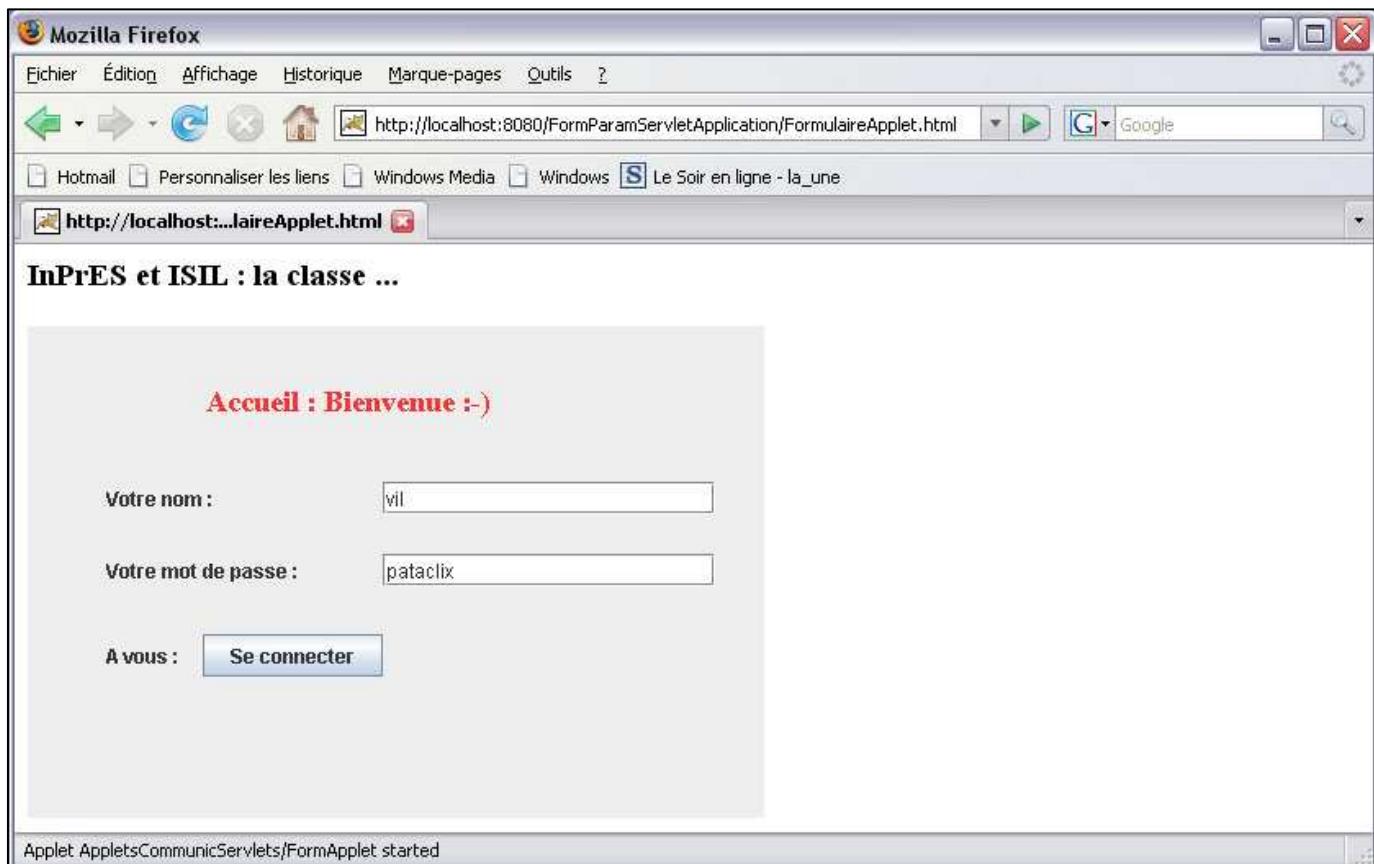
14.3 L'exécution de l'application Web

Notre applet sera intégrée dans une page HTML du type suivant :

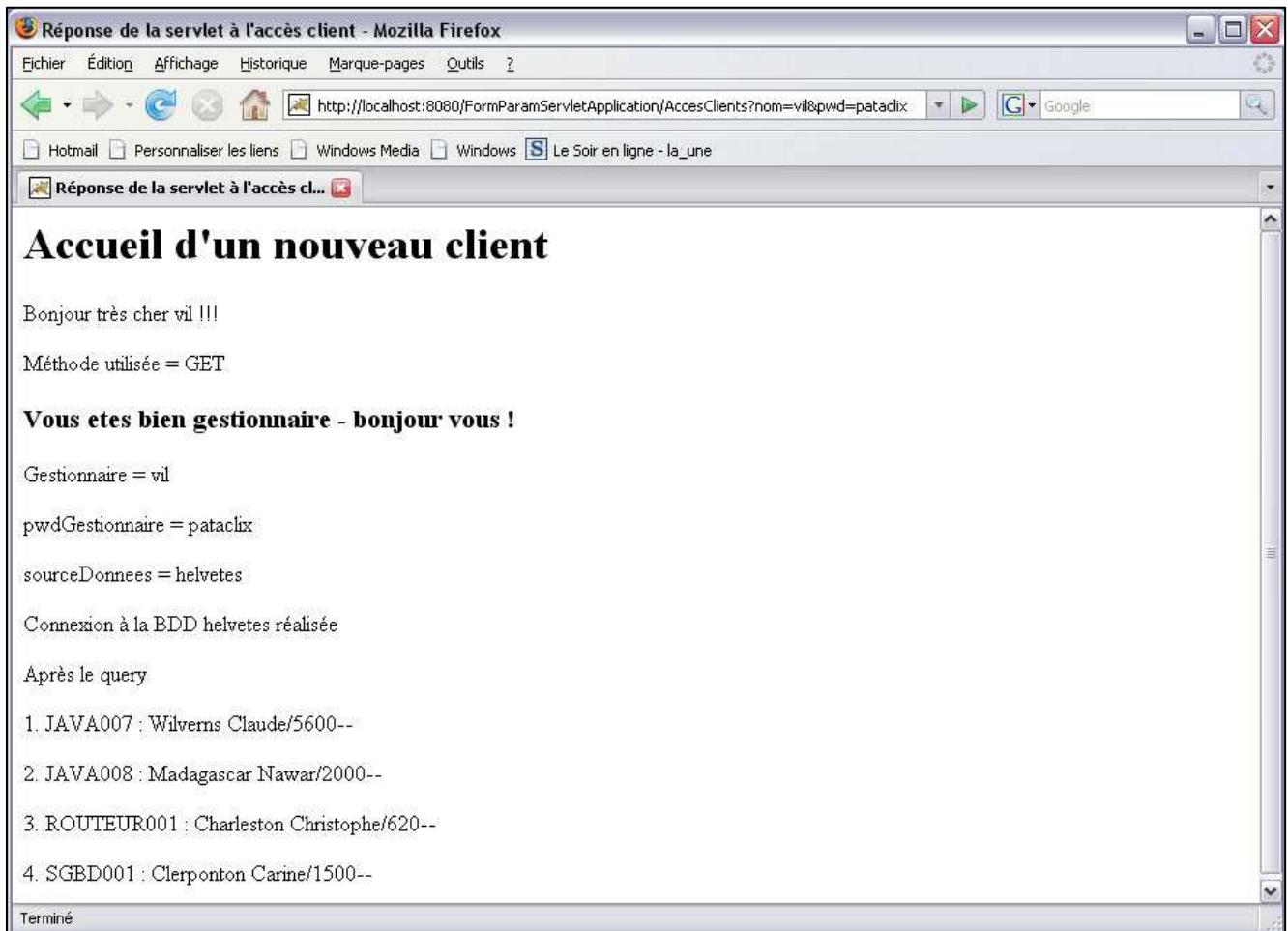
FormulaireApplet.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<title></title>
</head>
<h3>InPrES et ISIL : la classe ...</h3>
<body>
    <applet code="AppletsCommunicServlets/FormApplet.class"
            archive="AppletsForWebApplication.jar" width=450 height=300></applet>
</body>
</html>
```

Si nous attaquons par cette page :



l'appui sur le bouton "Se connecter" donnera :



14.4 Le monitoring HTTP avec NetBeans

Il est possible d'observer les requêtes HTTP occasionnées par nos développements Web lorsque nous les testons avec le serveur Tomcat intégré à NetBeans (pour autant que ce monitoring soit activé, ce qui est le cas par défaut : une case à cocher l'indique dans les Properties du Tomcat intégré [onglet Services]). Le contenu de ces requêtes est affiché dans l'onglet HTTP Monitor de la fenêtre inférieure gauche, un peu à la manière d'un sniffer évolué (mais sans les trames des protocoles des couches inférieures, évidemment, et sans les trames de réponse). Si nous reprenons l'application Web précédente, une exécution fournira les renseignements suivants. Pour la requête demandant la page HTML contenant l'applet :

Usages Output Search Results HTTP Monitor

All Records Current Records Request POST AccesClients [8:35 17/08/07]

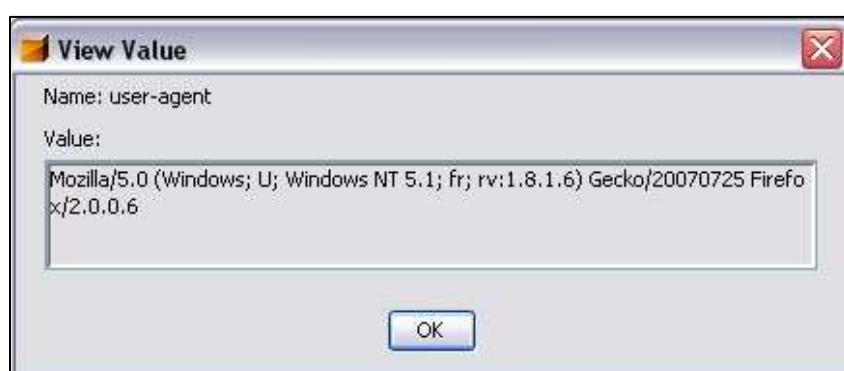
Request	
Request URI	/FormParamServletApplication/FormulaireAcces...
Method	GET
Query string	
Protocol	HTTP/1.1
Client IP address	127.0.0.1
Scheme	http
HTTP exit status (as set by servlet)	Could not be determined

There was no query string.

The screenshot shows the 'HTTP Monitor' interface with the 'Headers' tab selected. On the left, there's a tree view of records: 'All Records', 'Current Records' (expanded to show 'GET FormulaireAccesHelvetes.html [8:]' and 'POST AccesClients [8:35 17/08/07]'), and 'Saved Records'. The main pane shows a table of HTTP headers:

Header	Value	...
host	localhost:8084	...
user-agent	Mozilla/5.0 (Windows; U; Windows NT 5.1; fr; rv:1.8.1.6) Gecko/20070725 Firefox/2.0.0.6	...
accept	text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,/application/javascript;q=0.8,image/png,*/*;q=0.5	...
accept-language	fr,fr-fr;q=0.8,en-us;q=0.5,en;q=0.3	...
accept-encoding	gzip,deflate	...
accept charset	ISO-8859-1,utf-8;q=0.7,*;q=0.7	...
keep-alive	300	...
connection	keep-alive	...

Si les renseignements sont tronqués à l'affichage à cause de l'exigüité de la fenêtre, il suffit de cliquer sur la case adjacente contenant "..." pour obtenir un affichage complet (ici, pour "user-agent") :



Pour la requête sollicitant la servlet, cela donne :

The screenshot shows the 'HTTP Monitor' interface with the 'Request' and 'Headers' tabs selected for a specific POST request ('POST AccesClients [8:35 17/08/07]').

Request Tab:

Parameter	Value	...
Request URI	/FormParamServletApplication/AccesClients	...
Method	POST	...
Query string		...
Protocol	HTTP/1.1	...
Client IP address	127.0.0.1	...
Scheme	http	...
HTTP exit status (as set by servlet)	Could not be determined	...

Parameters Tab:

Parameter	Value	...
nom	vil	...
pwd	pataclix	...

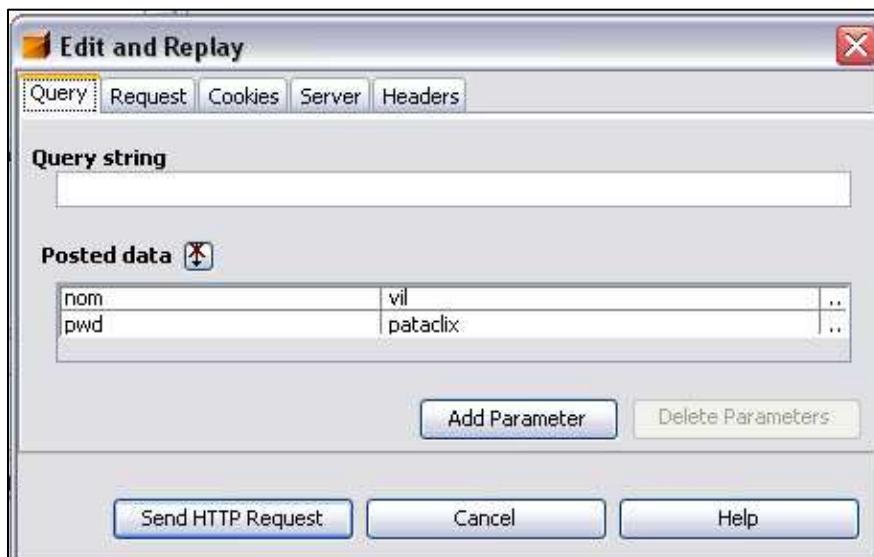
Headers Tab:

Header	Value	...
host	localhost:8084	...
user-agent	Mozilla/5.0 (Windows; U; Windows NT 5.1; fr; rv:1.8.1.6) Gecko/20070725 Firefox/2.0.0.6	...
accept	text/xml,application/xml,application/xhtml+xml;text/html;q=0.9;/application/javascript;q=0.8,image/png,*/*;q=0.5	...
accept-language	fr,fr-fr;q=0.8,en-us;q=0.5,en;q=0.3	...
accept-encoding	gzip,deflate	...
accept charset	ISO-8859-1,utf-8;q=0.7,*;q=0.7	...
keep-alive	300	...
connection	keep-alive	...
referer	http://localhost:8084/FormParamServletApplicatio...	...
content-type	application/x-www-form-urlencoded	...
content-length	20	...

On peut d'autres informations concernant les interlocuteurs comme :

The screenshot shows the 'HTTP Monitor' interface. On the left, there's a tree view with 'All Records', 'Current Records' (selected), and 'Saved Records'. Under 'Current Records', there are entries for 'GET FormulaireAccesHelvetes.html [8:35 17/08/07]' and 'POST AccesClients [8:35 17/08/07]'. The main pane is divided into 'Client' and 'Server' sections. The 'Client' section lists: Protocol (HTTP/1.1), Client IP address (127.0.0.1), Software used (Mozilla/5.0 (Windows; U; Windows NT 5.1; fr; rv:1...)), Locale(s) (fr,fr-fr;q=0.8,en-us;q=0.5,en;q=0.3), Encoding accepted (gzip,deflate), File formats accepted (text/xml,application/xml,application/xhtml+xml,te...), and Character sets accepted (ISO-8859-1,utf-8;q=0.7,*;q=0.7). The 'Server' section lists: Java version (2), Platform (Apache Tomcat/5.5.17), Hostname (localhost), and Port number of HTTP service (8084).

Mais surtout, on peut tester la servlet en modifiant ses paramètres. En effet, un clic droit sur la requête POST donne un menu permettant de relancer une nouvelle requête modifiée :



Les appuis sur "Ok" puis "Send HTTP Request" donnent cette fois :

Réponse de la servlet à l'accès client - Mozilla Firefox

Fichier Édition Affichage Historique Marque-pages Outils ?

http://localhost:8084/FormParamServletApplication/AccesClients?netbeans.replay=1187332519000&netbea

Hotmail Personnaliser les liens Windows Media Windows Le Soir en ligne - la_une

http://localhost:80... AccesHelvetes.html Réponse de la servlet à l'accès cl...

Accueil d'un nouveau client

Bonjour très cher vil !!!

Méthode utilisée = POST

Vous n'etes pas le gestionnaire - désolé

et une nouvelle requête est apparue dans le monitor :

Usages Output Search Results HTTP Monitor

All Records Current Records GET FormulaireAccesHelvetes.html [8:35] POST AccesClients [8:35 17/08/07] POST AccesClients [9:16 17/08/07]

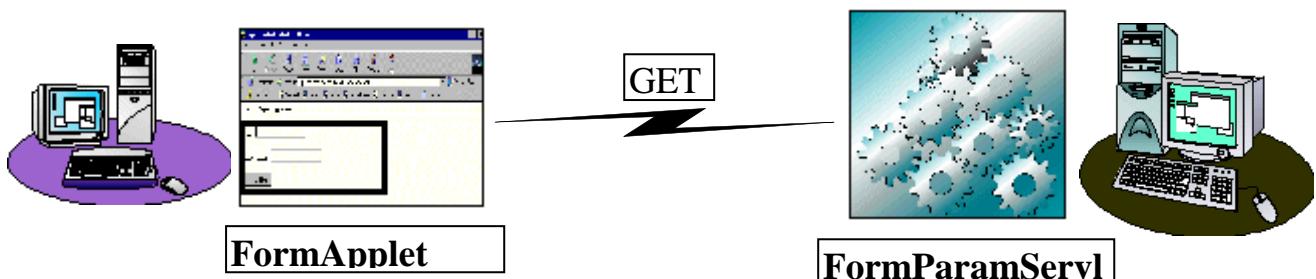
Request	
Request URI	/FormParamServletApplication/AccesClients
Method	POST
Query string	
Protocol	HTTP/1.1
Client IP address	127.0.0.1
Scheme	http
HTTP exit status (as set by servlet)	Could not be determined

Parameters

nom	vil
pwd	pataclac

14.5 L'utilisation d'une applet avec un serveur distant

Nous allons reprendre notre dialogue applet-servlet par formulaire interposé selon la méthode GET en ne travaillant plus en local, mais avec deux machines distinctes :



Evidemment, il nous y apporter quelques modifications pour l'applet. En effet, la construction de l'URL de la servlet va s'effectuer d'une manière différente afin d'être indépendant de la machine serveur – pour l'instant, elle invoque une servlet locale ☺ ! On pourrait taper l'URL à la dure dans un constructeur mais nous allons ici procéder différemment. Pour rappel, la méthode

public URL **getDocumentBase()**

renvoie un objet URL correspondant au chemin d'accès du répertoire de la page HTML utilisant l'applet. L'objet ainsi obtenu permet à son tour de récupérer le protocole, la machine hôte et le port au moyen des méthodes de la classe URL (évoquées dans le chapitre consacré

aux communications réseaux) **getProtocol()**, **getHost()** et **getPort()**. On peut dès lors créer l'URL de la servlet en usant du constructeur

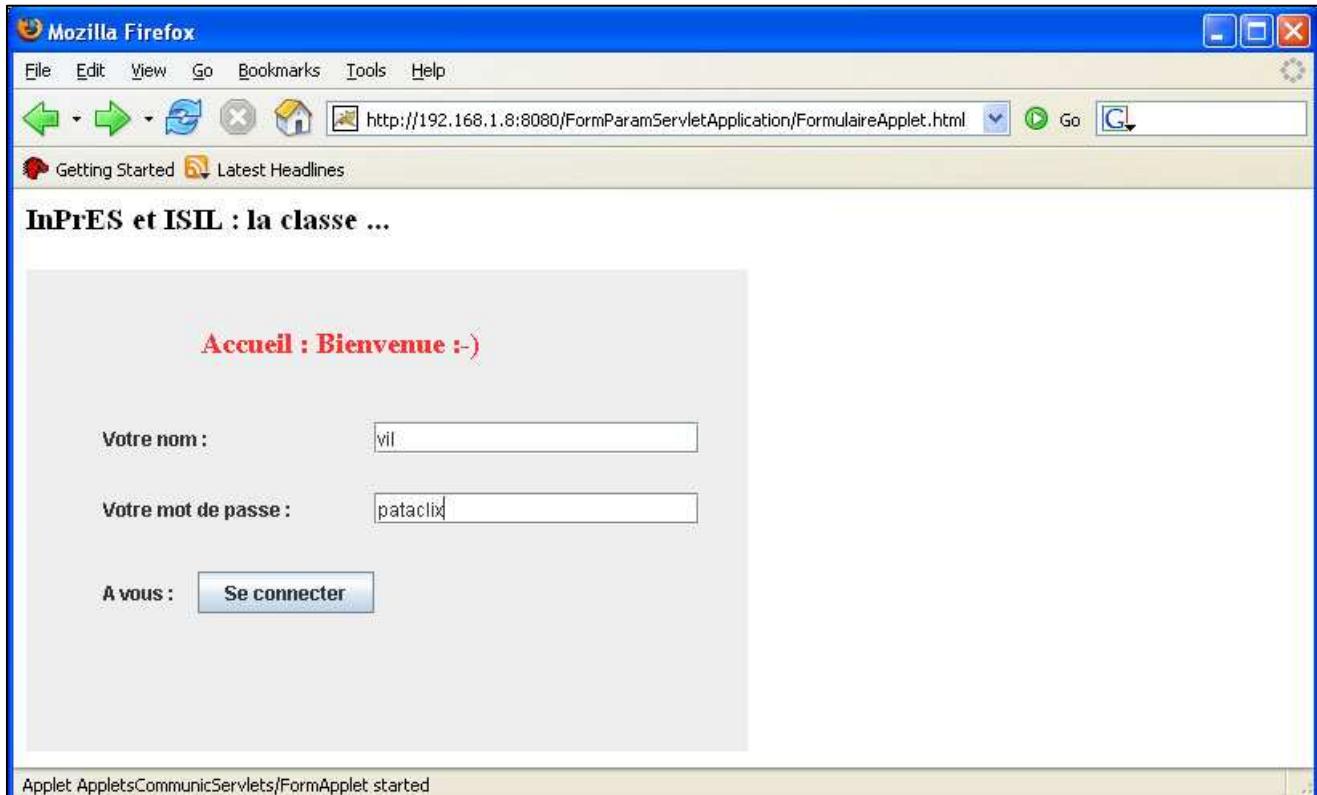
```
URL(String, String, int, String) // protocole, hôte, numéro de port, fichier
```

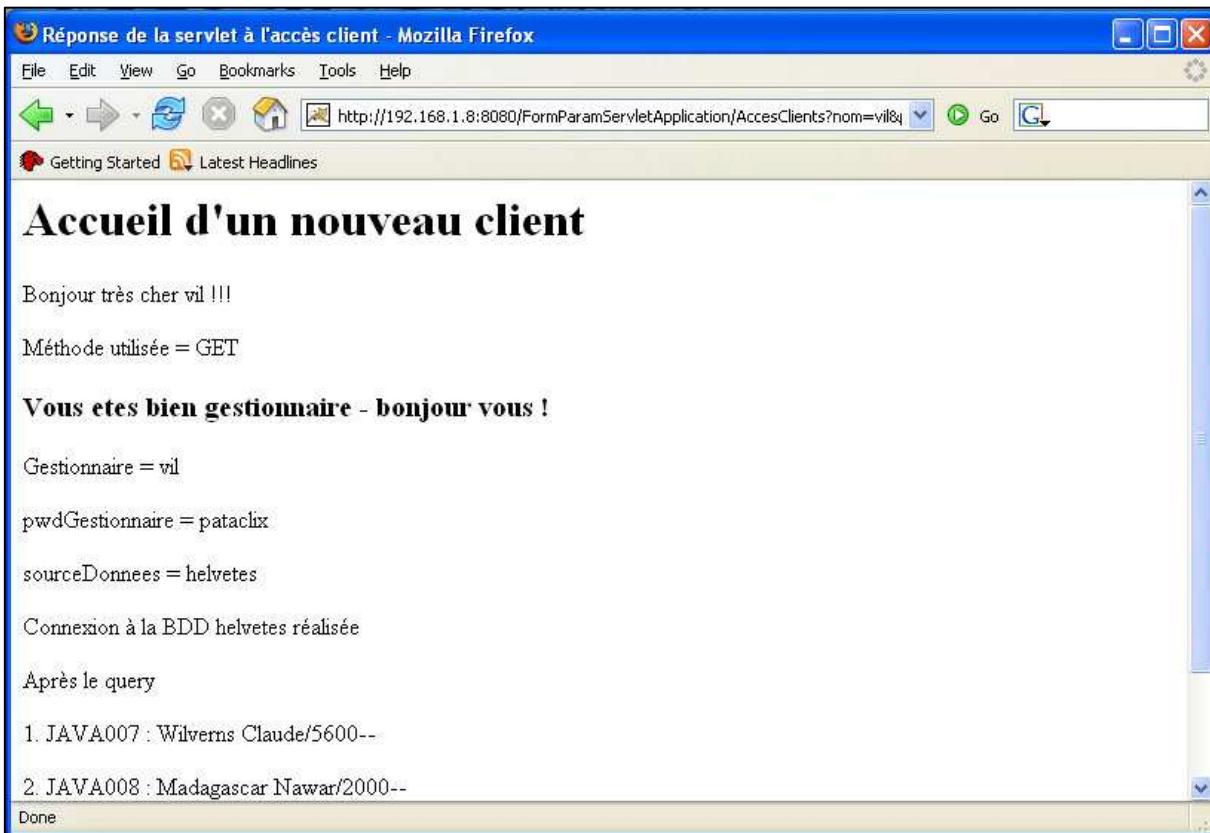
Ce qui nous donne :

FormApplet.java (modifications pour portabilité)

```
...
String adresseServlet = "/FormParamServletApplication/AccesClients"
    + "?nom=" + URLEncoder.encode(n) + "&prenom=" + URLEncoder.encode(p);
URL urlServlet = null;
try
{
    URL pageCourante = getDocumentBase();
    System.out.println("getDocu = " + pageCourante);
    String protocole = pageCourante.getProtocol();
    String machine = pageCourante.getHost();
    int port = pageCourante.getPort();
    System.out.println("p=" + protocole + " m=" + machine + " port=" + port);
    urlServlet = new URL (protocole, machine, port, adresseServlet);
}
catch (MalformedURLException ex)
{
    System.out.println("Aie aie - URL louche ! : " + ex.getMessage());
}
getAppletContext().showDocument(urlServlet);
...
```

Le reste de l'applet reste inchangé, tout comme la servlet. Une tentative d'exécution se passe sans problème :





14.6 Un environnement différent : Linux-Apache-Tomcat-MySQL

On peut aussi envisager d'utiliser une configuration Linux avec diverses configurations :

- ◆ Tomcat-MySQL (à l'InPrES : machines U2 et Indochine) – totalement similaire à la configuration Windows correspondante;
- ◆ Apache-Tomcat-MySQL (à l'InPrES : machine U2).

Si Tomcat nous est bien connu, les deux autres acteurs méritent peut-être quelques explications pour certains lecteurs ...

1) Le serveur Web **Apache** est sans doute le plus célèbre des serveurs Web. Il existe en fait pour différentes plates-formes (HPUnix, Digital Unix, Linux, Windows, Mac OsX). Il représente plus ou moins 2/3 du marché des serveurs Web. Dans la version utilisée ici, il est multiprocessus; il deviendra multithread dans sa version 2.0. Son architecture est modulaire : le noyau contient les fonctionnalités fondamentales tandis que les autres se trouvent dans des modules distincts qui peuvent être ajoutés de manière statique ou dynamique (comme les librairies traditionnelles). Apache est en écoute par défaut sur le port 80 (port http). Il s'exécute comme un service du démon inetd (ce dernier est un démon en écoute sur plusieurs ports, qui lance le programme correspondant à la requête reçue).

Un serveur que nous pourrons utiliser est donc u2.wildness.loc. Il gère en fait des **hôtes virtuels [virtual hosts]** : il s'agit de sites distincts se trouvant sur le même serveur. A l'heure actuelle, trois virtual hosts existent : http://prof.wildness.loc , http://u2.wildness.loc et http://student.wildness.loc. Chaque hôte doit posséder une entrée dans le DNS du serveur.

Un gros intérêt de cette façon de procéder est que l'on peut définir des directives au niveau d'un hôte virtuel ... Ainsi, chaque utilisateur s'est vu confié un compte correspondant à

une arborescence de sous-répertoires. Par exemple, l'utilisateur vilvens a accès à un répertoire racine "vilvens" qui comporte notamment :

- ◆ un sous-répertoire public_html : il contient la page index.html ainsi que les pages et applets créées par lui; les fichiers qui s'y trouvent sont destinés à être lus par tout le monde et doivent avoir le masque 755 (rwxr-xr-x);
- ◆ un sous-répertoire conf : on se doute que ce répertoire est dédié aux fichiers de configuration propres à l'utilisateur;
- ◆ un sous-répertoire servlets : à priori destiné aux servlets gérées par l'ancien moteur à servlets JServ.

Nom	Ext.	Taille	Date	Attr
[.hotjava]	<DIR>		31/05/2001 15:12:750	▲
[.seyon]	<DIR>		31/05/2001 15:12:750	
[.xfm]	<DIR>		31/05/2001 15:12:750	
[conf]	<DIR>		31/05/2001 15:15:750	
[public_html]	<DIR>		01/06/2001 07:25:750	
[servlets]	<DIR>		01/06/2001 07:14:750	
.bash_history		0	31/05/2001 15:12:640	
.bashrc		1 392	31/05/2001 15:12:640	
.dayplan		0	31/05/2001 15:12:640	
.dayplan	priv	0	31/05/2001 15:12:640	
.dvipsrc		208	31/05/2001 15:12:640	
.emacs		4 143	31/05/2001 15:12:640	
.exrc		1 124	31/05/2001 15:12:640	
.gimprc		5 376	31/05/2001 15:12:640	
.iazz		7 924	31/05/2001 15:12:640	
.kermrc		164	31/05/2001 15:12:640	
.lyxrc		10 376	31/05/2001 15:12:640	

2) Le monde Linux est très porté vers le gestionnaire de bases de données **MySQL**, qui est Open Source sous licence GPL (General Public Licence : le source est disponible et modifiable, il est intégrable dans toute application pourvu que celle-ci soit elle-même GPL). Nous avons déjà fait sa connaissance dans le chapitre consacré à JDBC (volume Java II).

Pour rappel, le SGBD est géré au moyen d'une métabase de données appelée logiquement "mysql"; cette base comporte 6 tables contenant les utilisateurs et les divers priviléges de ceux-ci sur les bases, les tables, les colonnes.

MySQL n'a pas toujours été pas un SGBD exemplaire ! Ainsi, un certain nombre de caractéristiques d'un SGBD classique (comme Oracle) étaient absentes dans les versions relativement récentes ☺ :

- ◆ pas de clés étrangères : plus précisément, elles ne sont pas gérées même si une clause "foreign key" est acceptée (ce gros défaut a disparu des versions récentes☺);
- ◆ pas de select imbriqués;
- ◆ pas de transactions (commit, rollback);
- ◆ pas de triggers;
- ◆ pas de procédures stockées (existent dans les versions récentes);
- ◆ pas de support des vues.

Un tel produit ne serait donc pas vraiment idéal pour enseigner les bases de données ... Par contre, on ne peut ignorer que, dans la foulée du développement de Linux, MySQL se répand à son tour. Pour ce qui nous concerne, dans le plus pur esprit Java, MySQL nous servira de

SGBD "neutre" (c'est-à-dire n'obligeant pas à adopter des solutions propriétaires) pour la gestion des données dans des tables plutôt que dans des fichiers, assez lourds à manipuler.

L'interface de gestion des bases de données sera ici **PHPMyAdmin**. Il s'agit, comme son nom l'indique, d'une interface construit en **PHP** (Personal Home Page), langage de script interprété côté serveur, essentiellement utilisé pour fournir des pages HTML dynamiques. L'interface en question permet de gérer facilement une base de données. On y accède, dans le cadre de l'infrastructure de l'école, depuis un browser par :

<http://u2.wildness.loc/> <http://phpmyadmin>

Après introduction du nom d'utilisateur et du mot de passe, on obtient la page de base :



Ici, l'utilisateur vilvens travaille dans sa base de donnée (également appelée "vilvens"). Normalement, il s'agira de créer une nouvelle table, par exemple une table clients. Il suffit pour cela de se déplacer en bas d'écran pour sélectionner :

A screenshot of a modal dialog box titled "Create new table on database vilvens:". It contains a list item "Name: clients" with a text input field, and "Fields: 4" with a "Go" button.

L'interface graphique permet de définir les différents champs de la table ainsi que leur propriétés (notamment les clés primaires). On obtient la génération automatique du script SQL :

```
CREATE TABLE clients (numClient VARCHAR (8) not null , nom VARCHAR (20) not null , prenom VARCHAR (20) not null , solde INT (11) not null , PRIMARY KEY (numClient))
```

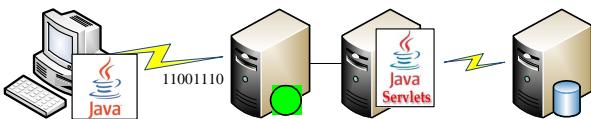
Une fois la table garnie de quelques tuples, on peut visionner ceux-ci par la commande browse ou, plus simplement encore, en cliquant sur l'icône précédent le nom de la table dans la frame de gauche :

The screenshot shows the phpMyAdmin interface in Microsoft Internet Explorer. The title bar reads "phpMyAdmin - Microsoft Internet Explorer". The left sidebar shows a tree view with "Home", "vilvens" (expanded), and "clients". The main content area is titled "Database vilvens - table clients". It displays a table with 5 records:

numClient	nom	prenom	solde
gen1	vilvens	claude	234532
gen2	mercenier	denys	555223
gem1	clermont	carine	2345
gem2	labelle	francesca	243543
gen3	charlet	christophe	837474

Below the table, there are navigation links: "Begin << Previous < > Show [30] rows starting from [] >> End". At the bottom of the content area, there is a link "Insert new row". The status bar at the bottom shows various icons and the text "zone Internet".

- C. Requête avec un formulaire en applet**
 - Réponse de la servlet en data ou objets

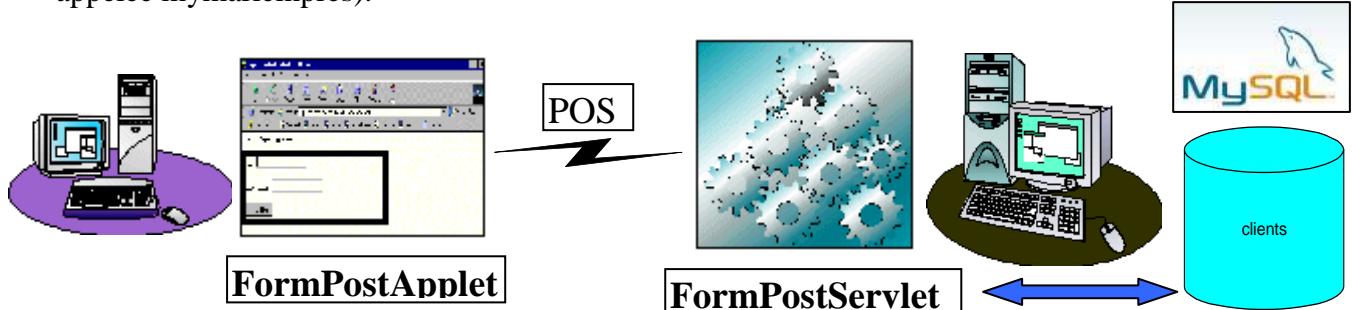


15. Une véritable communication applet-servlet : la méthode POST

15.1 Un tunnel http

On se souviendra que la méthode **POST** fonctionne selon le principe que les données envoyées au serveur servent d'entrée au programme de traitement des données, donc ici à la servlet, exactement comme si ces données étaient entrées au clavier. Il s'agit donc bien d'une redirection des entrées, accompagnée d'ailleurs d'une redirection des sorties. De plus, les paramètres issus du formulaire sont invisibles dans la zone URL du browser, à la différence de la méthode GET.

Nous allons donc construire un formulaire qui sera traité selon cette technique, cei dans un environnement Windows-MySQL. Disons, pour fixer les idées, qu'il permet à un client d'une chaîne de vente par e-commerce de s'inscrire. Le résultat sera donc qu'il se trouvera enregistré dans une base de données (table clients évoquée ci-dessus dans une base appelée mymarieinpres).



Visuellement, l'effet sera le suivant. La partie située au-dessus du bouton "Se connecter" correspond à la zone d'introduction de données tandis que la zone située en dessous contiendra la réponse de la servlet. :

*** Les joyeux commerçants ***	
Votre nom :	<input type="text"/>
Votre prénom :	<input type="text"/>
A vous !	Se connecter
Résultat de l'opération :	
<input type="text"/>	

Cette fois, l'applet va créer explicitement les deux flux de communication avec la servlet. La communication étant full-duplex (la méthode `doPost()` de la servlet attend deux flux), il faudra obligatoirement créer ces deux flux (même si l'on envisageait de se passer de la réponse de la servlet).

15.2 L'applet du tunnel http

La création de l'interface graphique de notre applet (appelons-la `FormPostApplet`) n'est évidemment pas un problème. Signalons cependant qu'il comporte deux `JTextField` (nom et prenom) et un `JTextArea` (resultat).

Par contre, il va nous falloir configurer notre connexion avec la servlet de manière précise et différente de celle par défaut. Une fois que nous aurons donc construit l'URL de la servlet de manière portable comme indiqué au point précédent, il nous faudra en extraire un objet `URLConnection` par :

```
URLConnection connexion = urlServlet.openConnection();
```

La configuration fera alors appel aux méthodes suivantes de la classe `URLConnection` :

- ◆ `public void setUseCaches(boolean usecaches)`
 - ◆ `public void setDefaultUseCaches(boolean defaultusecaches)`
- en passant false comme paramètre à ces méthodes pour empêcher le browser d'utiliser son cache (sans quoi le bouton reload devra être souvent sollicité !).
- ◆ `public void setDoOutput(boolean dooutput)`
- on signifie ici que l'on désire utiliser la connexion pour effectuer une sortie en positionnant le paramètre à true (il est à false par défaut !); bien sûr, la méthode sœur

```
public void setDoInput(boolean doinput)
```

existe avec une valeur par défaut de true.

D'autre part, il faut tenir compte du fait que la méthode POST nécessite la connaissance exacte du nombre de bytes à envoyer (champ "Content-Length" du header du protocole http). On pourrait penser à utiliser la longueur de la chaîne de caractères, mais ce serait oublier un peu vite la différence entre char et bytes en Java ! Nous allons donc plutôt utiliser ici un flux instance de la classe `ByteArrayOutputStream`. Comme son nom l'indique, ce flux, déjà rencontré dans le chapitre consacré à la cryptographie, représente une succession de bytes en mémoire; l'intérêt majeur d'un tel flux est que la quantité de bytes qu'il peut recevoir est indéterminée. On peut cependant fixer d'emblée une taille de base élevée si l'on sait très bien que ce sera nécessaire (par défaut, la capacité, extensible répétons-le, est de 32) en utilisant le constructeur :

```
public ByteArrayOutputStream(int size)
```

Nous construirons sur ce flux bas niveau un flux instance de la classe `PrintWriter`, flux orienté texte formaté, pour y confectionner notre requête. Cela donne :

```
ByteArrayOutputStream baos = new ByteArrayOutputStream(512);
```

```
PrintWriter pw = new PrintWriter(baos, true);      // true pour autoflush
String n = nom.getText();
String p = prenom.getText();
String infos = "nom=" + URLEncoder.encode(n)
               + "&prenom=" + URLEncoder.encode(p);
pw.print(infos);
pw.flush();    // pour être certain ;-)
```

Il nous reste alors, comme promis, à positionner les champs d'en-tête HTTP en utilisant la méthode d'URLConnection :

```
public void setRequestProperty(String key, String value)
```

où le premier paramètre permet de désigner un champ précis et le deuxième d'une donner la valeur. Soit ici :

```
String longueurEnvoyee = String.valueOf(baos.size());
connexion.setRequestProperty("Content-Length", longueurEnvoyee);
connexion.setRequestProperty("Content-Type", "application/x-www-form-urlencoded");
```

On peut enfin envoyer la requête en écrivant le contenu du flux-tableau sur le flux de sortie associé à la connexion. On utilise pour cela la méthode de ByteArrayOutputStream :

```
public synchronized void writeTo(OutputStream out) throws IOException
```

soit ici :

```
baos.writeTo(connexion.getOutputStream());
```

L'applet n'a plus alors qu'à ouvrir un canal d'entrée, à y lire les réponses et à les afficher dans le contrôle TextArea résultat. En résumé, voici l'applet (générée avec le concours de NetBeans) :

FormPostApplet.java
package TunnelApplet;
<pre>import java.net.*; import java.io.*; /** * @author Vilvens */ public class FormPostApplet extends javax.swing.JApplet { public void init() { ... } private void initComponents(){ ... } private void BConnecterActionPerformed(java.awt.event.ActionEvent evt) {</pre>

```

try
{
    /*A la dure :
    String adresseServlet =
        "http://localhost:8080/ServletsForTunnelHttp/FormPostServlet";
    URL urlServlet = new URL (adresseServlet);
    */
    String adresseServlet = "/ServletsForTunnelHttp/FormPostServlet";
        // nom du war et url de la servlet tel que définie dans web.xml
    URL pageCourante = getDocumentBase();
    System.out.println("getDocu = " + pageCourante);
    String protocole = pageCourante.getProtocol();
    String machine = pageCourante.getHost(); int port = pageCourante.getPort();
    System.out.println("p=" + protocole + " m=" + machine + " port=" + port);
    URL urlServlet = new URL (protocole, machine, port, adresseServlet);
    URLConnection connexion = urlServlet.openConnection();
    System.out.println("Connexion ouverte");
    connexion.setUseCaches(false);
    connexion.setDefaultUseCaches(false);
    connexion.setDoOutput(true);

    System.out.println("Ouverture permise");
    ByteArrayOutputStream baos = new ByteArrayOutputStream(512);
    PrintWriter pw = new PrintWriter(baos, true);
    System.out.println("Printwriter ok");

    String n = nom.getText(); String p = prenom.getText();
    String infos = "nom=" + URLEncoder.encode(n)
        + "&prenom=" + URLEncoder.encode(p);
    pw.print(infos); pw.flush();

    System.out.println("infos Printwriter ok");
    String longueurEnvoyee = String.valueOf(baos.size());
    connexion.setRequestProperty("Content-Length", longueurEnvoyee);
    connexion.setRequestProperty("Content-Type",
        "application/x-www-form-urlencoded");
    System.out.println("SetRequestProperty ok");
    baos.writeTo(connexion.getOutputStream());
    System.out.println("getOutputStream ok");

    BufferedReader entree = new BufferedReader(
        new InputStreamReader(connexion.getInputStream()));
    System.out.println("getInputStream ok");
    String reponse;
    while ((reponse=entree.readLine()) != null)
    {
        System.out.println("servlet> "+reponse);
        resultat.append(reponse);
        resultat.append("\n");
    }
}

```

```
        catch (MalformedURLException ex)
        {
            System.out.println("Aie aie - URL louche ! : " + ex.getMessage());
        }
        catch (IOException ex)
        {
            System.out.println("Aie aie - probleme ouverture connexion ! : " + ex.getMessage());
        }
    }

private javax.swing.JButton BConnecter;
...
private javax.swing.JScrollPane jScrollPane1;
private javax.swing.JTextField nom;
private javax.swing.JTextField prenom;
private javax.swing.JTextArea resultat;
}
```

A remarquer que l'applet écrit sur la console. Elle sera lancée dans la page HTML :

FormulairePostApplet.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<title></title>
</head>
<body>
<applet code="TunnelApplet/FormPostApplet.class"
archive="AppletsForTunnelHttp.jar" width=500 height=400></applet>
</body>
</html>
```

15.3 La servlet du tunnel http

Pour rappel, la servlet a pour rôle d'enregistrer le nouveau client dans la table clients de la base de données vilvens. Pour cela :

- ◆ elle se procure ses flux d'entrée et de sortie;
- ◆ précise qu'elle envoie une réponse en format text/text;
- ◆ se connecte à la base de données MySql (le jar avec le driver jdbc doit donc être monté);
- ◆ génère un nouveau numéro de client (de manière assez vague d'ailleurs et sans vérifier si ce client n'existe pas déjà – quelle sauvagerie ;-)...);
- ◆ insère le client dans la table.

Ce qui donne :

FormPostServlet.java

```
package TunnelServlet;

import java.io.*;
import java.net.*;
import java.sql.*;

import javax.servlet.*;
import javax.servlet.http.*;

/**
 * @author Vilvens
 */

public class FormPostServlet extends HttpServlet
{
    protected void processRequest (HttpServletRequest request, HttpServletResponse
        response) throws ServletException, IOException
    {
        response.setContentType("text/text");
        PrintWriter out = response.getWriter();
        out.println("Servlet en action !!!");
        String nomClient, prenomClient;
        nomClient = request.getParameter("nom");
        prenomClient = request.getParameter("prenom");

        Connection con;
        Statement instruc;
        ResultSet rs;
        int nbreClients=0;

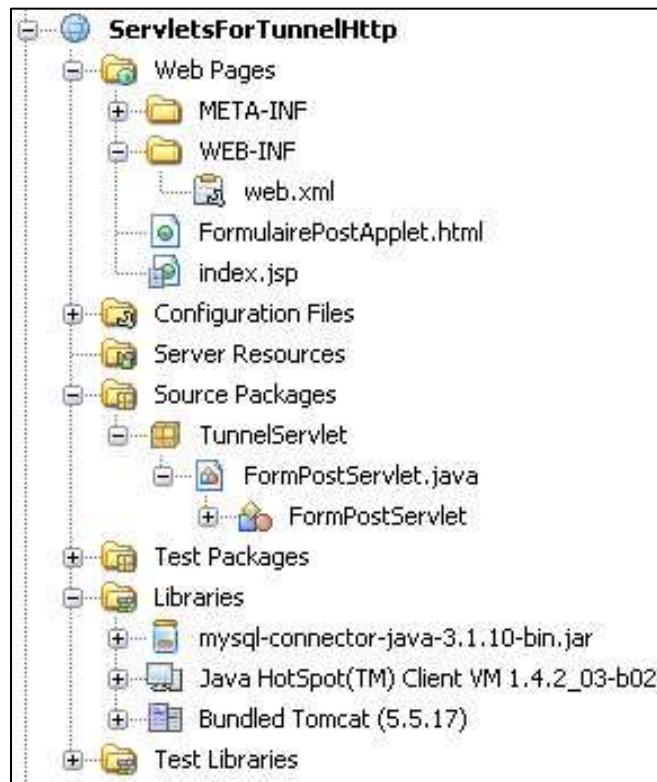
        try
        {
            Class.forName("com.mysql.jdbc.Driver");
        }
        catch (ClassNotFoundException e)
        {
            out.println("Driver MySQL non chargé " + e.getMessage());
        }
        try
        {
            con = DriverManager.getConnection
                ("jdbc:mysql://localhost:3306/mymarieinpres","vilvens","Banzai");
            out.println("Connexion à la BDD mymarieinpres réalisée");
            instruc = con.createStatement();
            rs = instruc.executeQuery("select * from clients");
            while (rs.next())
            {
                nbreClients++;
            }
        }
    }
}
```

```

        String nC = rs.getString("numClient");
        String n = rs.getString("nom");
        String p = rs.getString("prenom");
        int s = rs.getInt("solde");
        out.println(nbreClients + ". " + nC + " : " + n + " " + p + "/" + s);
    }
    out.println("Il y a : " + nbreClients + " clients");
    int numNouv = nbreClients+1;
    String numNouvClient = String.valueOf(nbreClients);
    String numClient;
    numClient = new String("JAVA"+numNouvClient);
    out.println("Numéro de client généré = " + numClient);
    int soldeParDefaut = 100;
    String requete = "insert into clients (numClient, nom, prenom, solde) values (" +
        + """" + numClient + """",
        + """" + nomClient + """",
        + """" + prenomClient + """",
        + """" + soldeParDefaut + ")");
    instruc.executeUpdate(requete);
}
catch (SQLException e)
{
    out.println("Erreur JDBC-OBDC : " + e.getMessage());
}
out.close();
}
protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException
{
    processRequest(request, response);
}
protected void doPost(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException
{
    processRequest(request, response);
}
public String getServletInfo()
{
    return "Servlet pour tunnel HTTP";
}
}

```

Le projet de la servlet ressemble donc à ceci :



15.4 Le test du tunnel HTTP

Un exemple d'exécution en local sera donc :

The screenshot shows a Mozilla Firefox browser window. The address bar displays the URL `http://localhost:8080/ServletsForTunnelHttp/FormulairePostApplet.html`. The page content is as follows:

*** Les joyeux commerçants ***

Votre nom :

Votre prénom :

A vous !

Résultat de l'opération :

Servlet en action !!!
Connexion à la BDD mymarieinpres réalisée
1. JABD001 : Kuty Ludovic/50000
2. JAVA007 : Vilvens Claude/8900
3. JAVA008 : Madani Nawar/7400

Applet TunnelApplet/FormPostApplet started

La console Java du browser affiche bien les traces de l'insertion d'un nouveau client :

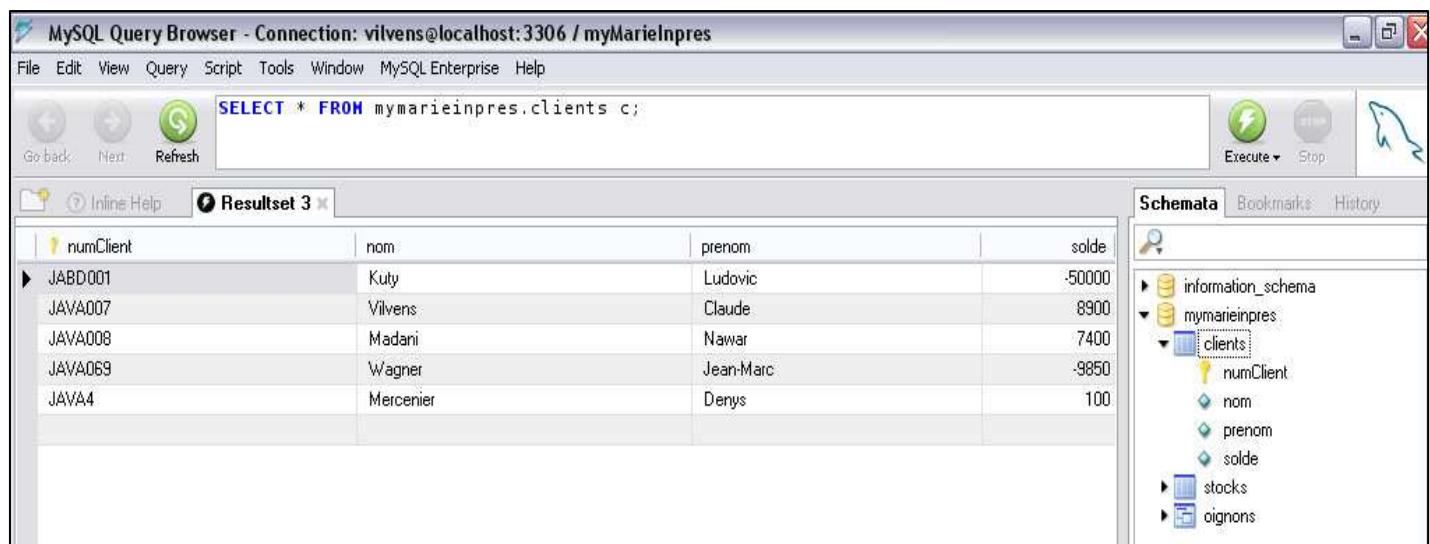


The screenshot shows a window titled "Console Java" with the following log output:

```
Connexion ouverte
Ouverture permise
Printwriter ok
infos Printwriter ok
SetRequestProperty ok
getOutputStream ok
getInputStream ok
servlet> Servlet en action !!!
servlet> Connexion à la BDD mymarieinpres réalisée
servlet> 1. JABD001 : Kuty Ludovic/-50000
servlet> 2. JAVA007 : Vilvens Claude/8900
servlet> 3. JAVA008 : Madani Nawar/7400
servlet> 4. JAVA069 : Wagner Jean-Marc/-9850
servlet> Il y a : 4 clients
servlet> Numéro de client généré = JAVA4
```

At the bottom of the window are three buttons: "Effacer" (Clear), "Copier" (Copy), and "Fermer" (Close).

On peut vérifier que "Mercenier" a bien été ajouté dans la base de données :



The screenshot shows the MySQL Query Browser interface. A query has been run:

```
SELECT * FROM mymarieinpres.clients c;
```

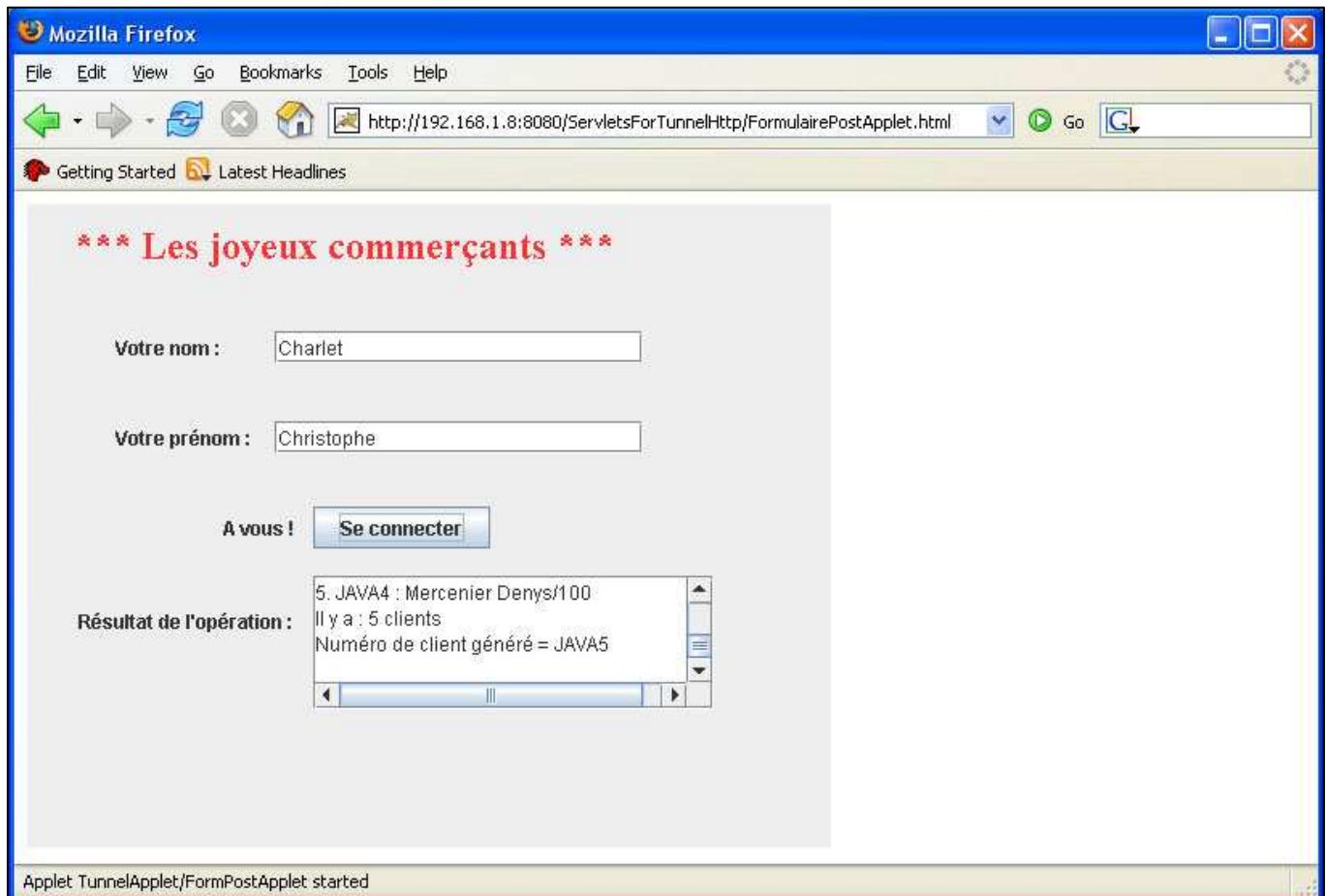
The results are displayed in a table titled "Resultset 3". The table has four columns: numClient, nom, prenom, and solde. The data is as follows:

numClient	nom	prenom	solde
JABD001	Kuty	Ludovic	-50000
JAVA007	Vilvens	Claude	8900
JAVA008	Madani	Nawar	7400
JAVA069	Wagner	Jean-Marc	-9850
JAVA4	Mercenier	Denys	100

On the right side, there is a "Schema" tree view showing the database structure:

- information_schema
- mymarieinpres
 - clients
 - numClient
 - nom
 - prenom
 - solde
 - stocks
 - oignons

Un second test à partir d'une machine distante donnera :



et la Java Team de l'InPrES-ISIL est ainsi au complet :

numClient	nom	prenom
JABD001	Kuly	Ludovic
JAVA007	Vilvens	Claude
JAVA008	Madani	Nawar
JAVA069	Wagner	Jean-Marc
JAVA4	Mercenier	Denys
JAVA5	Charlet	Christophe

Remarque

Avant de travailler sur un serveur distant, on teste souvent en local. Attention cependant ! Les fautes classiques lors du test d'un tunnel en local sont :

- 1) lancer l'exécution de la servlet directement : forcément, il y aura erreur, car la servlet ne renvoie pas une page mais essaie d'écrire dans un pipe qui n'existe pas ...
- 2) lancer l'exécution de l'applet ou charger la page html par quelque chose du genre :

file:///c:/java-sun-application/TunnelHttp/FormPostServlet.html

- l'applet ne parvient pas à se connecter à la servlet, car elle a été chargée localement; donc, elle ne peut se connecter qu'à la machine dont elle provient, soit le localhost et pas la machine claude visée !

15.5 Du point de vue du trafic réseau

On peut observer le tunnel http au moyen d'un sniffer. Ainsi, la requête client implique en fait deux requêtes HTTP (une pour la page, une pour le fichier class de l'applet) :

1 | 17:49:22.780 | 192.168.2.1:1039 | 192.168.2.2:8081 | TCP | 10 | 17:50:2

Time Offset	Pac...	Dat...	Data
→ 17:49:22.780	62	0	
← 17:49:22.780	62	0	
→ 17:49:22.780	60	0	
→ 17:49:22.780	487	433	GET /FormPostApplet.html HTTP/1.1 Accept: image/g...
← 17:49:22.795	54	0	
← 17:49:23.795	170	116	HTTP/1.1 304 Not Modified Date: Mon, 21 Jun 2004 ...
→ 17:49:23.795	60	0	
← 17:50:23.639	54	0	
→ 17:50:23.639	60	0	

3 | 17:49:29.874 | 192.168.2.1:1040 | 192.168.2.2:8081 | TCP | 11 | 17:49:4

Time Offset	Packe...	Data...	Data
→ 17:49:29.874	62	0	
← 17:49:29.874	62	0	
→ 17:49:29.874	60	0	
→ 17:49:30.874	347	293	GET /FormPostApplet.class HTTP/1.1 Cache-Control:...
← 17:49:30.874	54	0	
← 17:49:31.874	170	116	HTTP/1.1 304 Not Modified Date: Mon, 21 Jun 2004 ...
→ 17:49:31.874	60	0	
→ 17:49:42.014	60	0	
← 17:49:42.014	54	0	

L'appel de la servlet proprement dite provoque bien une réponse supportée par une trame HTTP :

6 | 17:50:54.092 | 192.168.2.1:1041 | 192.168.2.2:8081 | TCP | 13 | 17:51:0

Time Offset	Packe...	Data...	Data
→ 17:50:54.092	62	0	
← 17:50:54.092	62	0	
→ 17:50:54.092	60	0	
→ 17:50:54.092	370	316	POST /servlet/FormPostServlet HTTP/1.1 Content-Le...
→ 17:50:54.092	88	34	nom=charmant&prenom=sophie&sexe=2
← 17:50:54.092	54	0	
← 17:50:55.108	213	159	HTTP/1.1 200 OK Content-Type: text/html Date: Mo...
← 17:50:55.108	849	795	150 Servlet en action !!! Connexion à ...
→ 17:50:55.108	60	0	
→ 17:51:05.249	60	0	
← 17:51:05.249	54	0	
← 17:51:05.249	54	0	
→ 17:51:05.249	60	0	

15.6 Le modèle multithread

Il est sans doute temps de se souvenir d'une caractéristique essentielle du modèle des servlets, caractéristique qui marque une différence fondamentale avec les CGIs : une servlet n'est instanciée qu'une seule fois. *Chaque nouveau client qui s'y connecte est pris en charge par un thread distinct* et tous ces threads travaillent sur le même objet servlet. Celui-ci doit donc, pour les ressources non partageables de manière triviale, jouer le rôle d'un **moniteur**. Autrement dit, le code sensible, comme par exemple celui-ci d'accès à une base de donnée, se doit de se trouver dans une méthode ou un bloc **synchronized**.

Nous pouvons donc écrire la servlet précédente en utilisant de telles méthodes (cette fois, pas de code généré, rien que du "from scratch" ;-)) :

FormPostSyncServlet.java

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.sql.*;

public class FormPostSyncServlet extends HttpServlet
{
    Connection con;
    Statement instruc;
    ResultSet rs;
    public void doPost(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException
    {
        resp.setContentType("text/text");
        PrintWriter sortie = resp.getWriter();
        sortie.println("Servlet en action !!!");
        String nomClient, prenomClient;
        nomClient = req.getParameter("nom");
        prenomClient = req.getParameter("prenom");

        int nbreClients=0;
        try
        {
            Class.forName("com.mysql.jdbc.Driver");
            //Class.forName("org.gjt.mm.mysql.Driver");
        }
        catch (ClassNotFoundException e)
        {
            sortie.println("Driver MySQL non chargé " + e.getMessage());
        }

        try
        {
            con = DriverManager.getConnection
                ("jdbc:mysql://localhost:3306/mymarieinpres","vilvens","GrosZZ");
            sortie.println("Connexion à la BDD vilvens réalisée");
            instruc = con.createStatement();
        }
```

```

synchronized (this)
{
    rs = instruc.executeQuery("select * from clients");
    while (rs.next())
    {
        nbreClients++;
        String nC = rs.getString("numClient");
        String n = rs.getString("nom");
        String p = rs.getString("prenom");
        int s = rs.getInt("solde");
        sortie.println(nbreClients + ". " + nC + " : " + n + " " + p + "/" + s);
    }
    sortie.println("Il y a : " + nbreClients + " clients");
    int numNouv = nbreClients+1;
    String numNouvClient = String.valueOf(numNouv);
    String numClient;
    if (sexeClient.equals("1"))numClient = new String("gen"+numNouvClient);
    else numClient = new String("gem"+numNouvClient);
    sortie.println("Numéro de client généré = " + numClient);
    int soldeParDefaut = 100;
    String requete = "insert into clients (numClient, nom, prenom, solde) values (" +
        """" + numClient + """",
        """" + nomClient + """",
        """" + prenomClient + """",
        """" + soldeParDefaut + ")";
    instruc.execute(requete);
}
catch (SQLException e)
{
    sortie.println("Erreur JDBC-OBDC : " + e.getMessage());
}
sortie.close();
}

/*
A prévoir :
private synchronized boolean SyncSqlSelect (String requete) throws SQLException
private synchronized boolean SyncSqlInsert (String requete) throws SQLException
*/
}

```

Remarquons, en passant, l'utilisation de la méthode de l'interface Statement :

public abstract boolean **execute**(String sql) throws SQLException

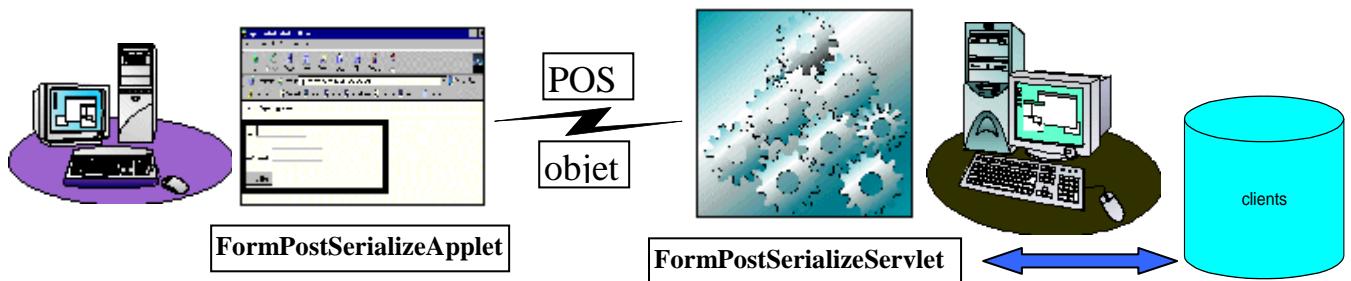
dont la valeur la valeur renvoyée est true si il y a au moins un ResultSet obtenu comme résultat de la requête.

16. La communication applet-servlet par objets sérialisés

16.1 Un package global

Une autre manière de faire communiquer une applet avec une servlet est d'utiliser des objets sérialisés, tout comme dans une communication classique par sockets :

- ◆ l'applet sérialise des objets sur des flux Object[In|Out]putStream obtenus depuis un objet URLConnection conduisant à la servlet;
- ◆ la servlet sérialise sur des flux Object[In|Out]putStream obtenus respectivement depuis ses objets HttpServletRequest req et HttpServletResponse.



Les classes des objets échangés se doivent, comme on sait, d'implémenter l'interface Serializable. De plus, ces classes doivent manifestement être connues tant de l'applet que de la servlet. On pense évidemment immédiatement à placer toutes ces classes dans un package, disons FormPostSerialize.

Reprendons le contexte des exemples précédents (mais cette fois dans le contexte d'un serveur Web Linux – ici, la machine U2 avec l'hôte virtuel prof.wildness.loc) et imaginons que l'applet permet à un utilisateur déjà enregistré d'obtenir son numéro de client et son solde actuel. Les classes dont les instances seront échangées entre les deux acteurs semblent évidentes (elles sont minimalistes, les variables membres sont publiques dans le seul but de ne pas traîner ;-)) :

FormPostSerializeClient.java

package *FormPostSerialize*;

```

import java.io.*;

class clientRequete implements Serializable
{
    public String nomClient;
    public String prenomClient;
    public String sexeClient;

    public clientRequete (String n, String p, String s)
    {
        nomClient=n; prenomClient=p; sexeClient=s;
    }
}
  
```

```
class clientReponse implements Serializable
{
    public String numClient;
    public String nomClient;
    public String prenomClient;
    public int soldeClient;

    public clientReponse (String nu, String no, String p, int s)
    {
        numClient=nu; nomClient=no; prenomClient=p; soldeClient=s;
    }
}
```

16.2 L'applet de la communication sérialisée

Elle est très semblable à celle du tunnel HTTP, si ce n'est que :

- ◆ elle instancie des objets qu'elle [dé]sérialise avec les méthodes [read|write]Object();
- ◆ le champ Content-Type du header HTTP est cette fois "application/x-java-serialized-object".

L'applet s'écrit donc (sans générateur) :

FormPostSerializeApplet.java

```
package FormPostSerialize;

import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.net.*;
import java.io.*;

public class FormPostSerializeApplet extends Applet implements ActionListener
{
    TextField nom, prenom, sexe;
    TextArea resultat;
    public void init()
    {
        resize(400,200); setLayout(new GridLayout(6,2));
        add(new Label("/** Les joyeux commerçants **"));
        add(new Label("Veuillez entrer vos coordonnées ..."));
        add(new Label ("Nom : ")); nom = new TextField("",20);add(nom);
        add(new Label ("Prenom : ")); prenom = new TextField("",20);add(prenom);
        add(new Label ("Sexe (1 ou 2) : ")); sexe = new TextField("",1);add(sexe);
        add(new Label("A vous !!!"));
        Button seConnecter = new Button ("se connecter"); add(seConnecter);
        add(new Label ("Résultat opération : : "));
        resultat = new TextArea("", 10, 30, TextArea.SCROLLBARS_BOTH);add(resultat);
        seConnecter.addActionListener(this);
    }
```

```

public void actionPerformed(ActionEvent e)
{
    if (e.getActionCommand() == "se connecter")
    {
        try
        {
            String adresseServlet =
                "http://prof.wildness.loc/vilvens/servlet/FormPostSerialize.FormPostSerializeServlet";
            URL urlServlet = new URL (adresseServlet);
            URLConnection connexion = urlServlet.openConnection();
            System.out.println("Connexion ouverte");
            connexion.setUseCaches(false); connexion.setDefaultUseCaches(false);
            connexion.setRequestProperty("Content-Type",
                "application/x-java-serialized-object");
            connexion.setDoOutput(true);
            System.out.println("Ouverture permise");

            ObjectOutputStream oos =
                new ObjectOutputStream(connexion.getOutputStream());

            String n = nom.getText();String p = prenom.getText();String s = sexe.getText();
            clientRequete cli = new clientRequete (n,p,s);
            oos.writeObject(cli);oos.flush();

            ObjectInputStream ois = new ObjectInputStream(connexion.getInputStream());
            String reponse = (String)ois.readObject();
            resultat.append(reponse);resultat.append("\n");
            if (reponse.equals("Client OK"))
            {
                clientReponse cliR = (clientReponse)ois.readObject();
                resultat.append("numero attribue =" +cliR.numClient);resultat.append("\n");
                resultat.append("solde par defaut =" + new Integer(cliR.soldeClient).toString());
                resultat.append("\n");
            }
            else resultat.append("Réponse servlet = " + reponse + "\n");
        }
        catch (MalformedURLException ex)
        { System.out.println("Aie aie - URL louche ! : " + ex.getMessage()); }
        catch (ClassNotFoundException ex)
        { System.out.println("Aie aie - definition classe inconnue ! : " + ex.getMessage()); }
        catch (IOException ex)
        { System.out.println("Aie aie - probleme ouverture connexion ! : " + ex.getMessage() );
        }
    }
}
}

```

On remarquera que l'on use du fait que la classe String implémente Serializable !

16.3 La servlet de la communication sérialisée

Elle reçoit donc un objet instanciant la classe client Requete, recherche les nom et prénom associés dans la base de données et, en cas de recherche fructueuse, construit un objet clientReponse contenant le numéro de client et le solde.

FormPostSerializeServlet.java

```
package FormPostSerialize;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.sql.*;

public class FormPostSerializeServlet extends HttpServlet
{
    public void doPost(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException
    {
        resp.setContentType("application/x-java-serialized-object");
        ObjectOutputStream oos = new ObjectOutputStream(resp.getOutputStream());
        ObjectInputStream ois = new ObjectInputStream(req.getInputStream());
        ClientRequete cliR;

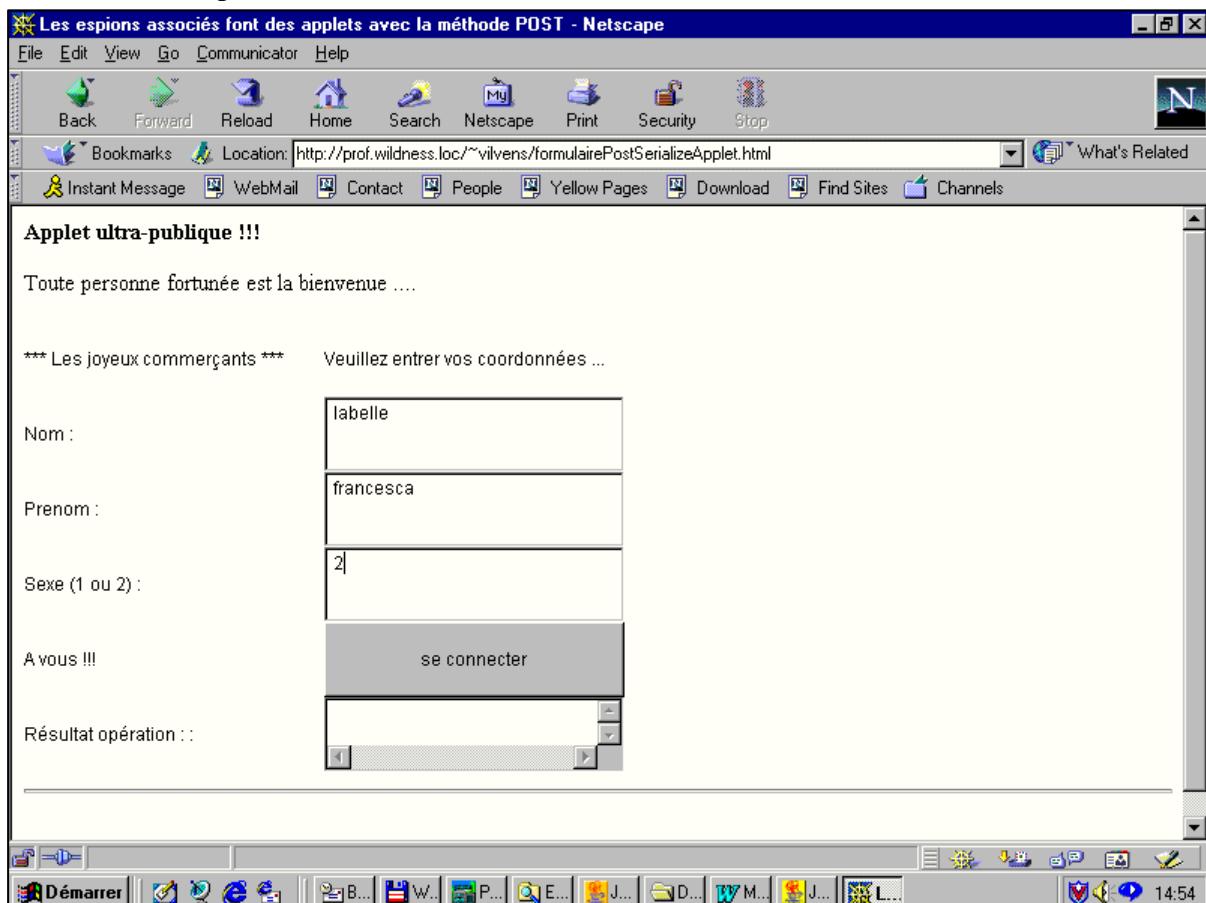
        String nomClient=null, prenomClient=null, sexeClient;
        try
        {
            cliR = (ClientRequete)ois.readObject();
            nomClient = cliR.nomClient;
            prenomClient = cliR.prenomClient;
            sexeClient = cliR.sexeClient;
        }
        catch (ClassNotFoundException ex)
        {oos.writeObject("Aie aie - definition classe inconnue ! : " + ex.getMessage());oos.flush();}

        Connection con;
        Statement instruc;
        ResultSet rs;
        int nbreClients=0;
        try
        {
            Class.forName("org.gjt.mm.mysql.Driver"); // autre driver MySQL
        }
        catch (ClassNotFoundException e)
        {oos.writeObject("Driver MySQL non chargé " + e.getMessage()); }

        try
        {
            con = DriverManager.getConnection("jdbc:mysql://localhost/vilvens","vilvens","xxxx");
            instruc = con.createStatement();
        }
```

```
rs = instruc.executeQuery(  
    "select * from clients where nom='"+ nomClient+" and prenom = ""  
    + prenomClient + "");  
  
if (rs.next())  
{  
    String numClient = rs.getString(1);  
    int soldeClient = rs.getInt(4);  
    clientReponse cli = new clientReponse(numClient, nomClient, prenomClient,  
        soldeClient);  
    oos.writeObject("Client OK"); oos.flush();  
    oos.writeObject(cli);  
}  
else oos.writeObject("Pas trouvé dans la base de données");  
  
}  
catch (SQLException e)  
{ oos.writeObject("Erreur JDBC-OBDC : " + e.getMessage()); }  
}  
}
```

Un exemple d'exécution serait :



On obtient dans la zone de texte de l'applet :

Client OK

numero attribue = gem2
solde par defaut = 243543

Si l'on tente un essai pour "labelle louisa" :

Pas trouvé dans la base de données

Réponse servlet = Pas trouvé dans la base de données

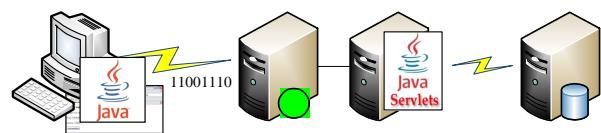
et pour "christophe charlet" :

Client OK

numero attribue = gen3
solde par defaut = 837474

D. Applications et sessions Web

- Plusieurs servlets qui communiquent



17. Les cookies

17.1 Le principe

Un cookie est une information (plus précisément un fichier texte) envoyée à un browser par un serveur qui vient d'être accédé afin que cette information soit sauvegardée sur le client.

Chaque fois que le browser s'adressera à ce serveur, il lui renverra le contenu du cookie. De cette manière, un serveur peut, par exemple, identifier d'une manière unique un client, puisque c'est le serveur lui-même qui a créé l'identification qu'il récupère.

Les cookies sont notamment utilisés pour conserver des données d'une visite d'un site à une autre, pour identifier l'utilisateur et éventuellement l'authentifier (dans le contexte du e-commerce, on parle encore de "*shopping cart*" style applications), pour enregistrer la date de sa dernière visite, etc. Ils ont été parfois décriés, car il faut que le client laisse sa garde pour laisser le cookie s'enregistrer. Cependant, ils sont devenus un standard de fait, objet de la RFC 2109.

Les cookies ont été introduits pour la première fois dans Netscape Navigator. Un cookie peut y avoir un contenu quelconque. Par contre, dans le cas d'Internet Explorer, un cookie doit toujours se terminer par '\n' ...

17.2 La création des cookies en Java

Les cookies sont supportés par le protocole HTTP, qui tient compte de la nature particulière de ceux-ci dans les champs de ses headers. Cependant, en pratique et comme d'habitude, cet aspect de la réalité est encapsulé dans les méthodes fournies par le package javax.servlet.http. Ainsi, un cookie sera ici une instance de la classe **Cookie**. Comme un cookie est caractérisé par un nom et une valeur (c'est-à-dire le texte qui lui est associé), son constructeur est très logiquement :

```
public Cookie(String name, String value)
```

Ainsi, si nous souhaitons par exemple, dans le dialogue applet-servlet du paragraphe 13.3, mémoriser la date de la dernière visite et attribuer un identifiant de client, nous créerons deux cookies :

```

Date dateDerVisite = new Date();
String dateDerVisiteTexte = DateFormat.
    getDateTimeInstance(DateFormat.FULL,DateFormat.FULL,Locale.FRANCE).
    format(dateDerVisite);
String dateDerVisiteTextePourCookie = dateDerVisiteTexte + "\n";
    // pour Internet Explorer
Cookie cookieVisite = new Cookie ("derVisite", dateDerVisiteTextePourCookie);

// comptage du nombre de clients existants
Cookie cookieId = new Cookie ("id", cpt);

```

17.3 Ecrire et lire des cookies

On s'en doute un peu, puisque les cookies sont supportés par HTTP, les paramètres **HttpServletRequest** et **HttpServletResponse** des méthodes `doGet()` et `doPost()` possèdent des méthodes de lecture et d'écriture de cookies :

- ♦ les cookies sont écrits au moyen de la méthode :

```
public abstract void addCookie(Cookie cookie)
```

- ♦ les cookies sont lus au moyen de la méthode :

```
public abstract Cookie[] getCookies()
```

Cette dernière renvoie un tableau contenant tous les cookies fournis par le browser dans sa requête. On peut alors le parcourir pour rechercher un cookie bien précis; bien sûr, l'obtention d'une référence nulle pour un cookie cherché signifie qu'il n'existe pas. Par contre, pour tout cookie existant, on peut récupérer son nom et sa valeur par les méthodes de la classe `Cookie` :

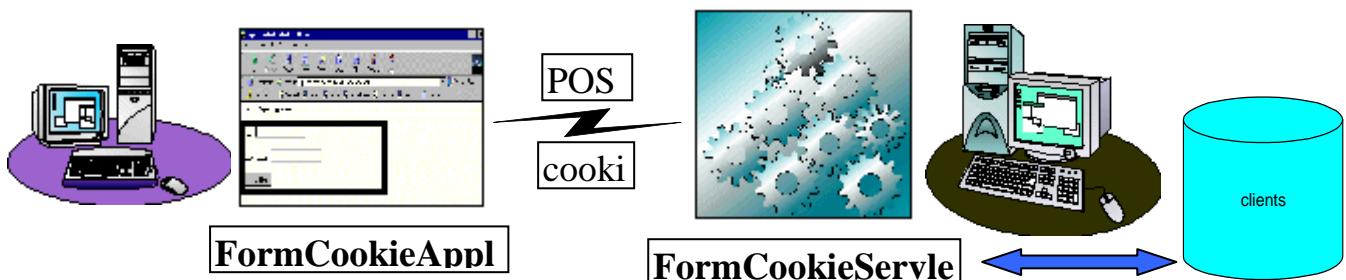
```

public String getName()
public String getValue()

```

17.4 Un cookie pour la dernière visite

Un exemple classique de cookie persistant est de mémoriser la date de la dernière visite au serveur. C'est ce que fait la servlet ci-dessous. Un cookie `cookieDerVisite` permet d'enregistrer la date courante (au format local).



Une subtilité : un cookie ne peut contenir d'espaces ! On a donc remplacé de tels espaces par un point au moyen de la méthode

```
public String replace(char oldChar, char newChar)
```

de la classe String. On remarquera aussi, dans le contexte de la récupération de la date courante, la qualification complète de la classe Date (java.util.Date) afin d'éviter la confusion avec la classe Date du package sql !

FormCookieServlet.java (version cookieDerVisite)

```

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.sql.*;
import java.util.*;
import java.text.*;

public class FormCookieServlet extends HttpServlet
{
    Connection con;
    Statement instruc;
    ResultSet rs;
    public void doPost(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException
    {
        java.util.Date dateDerVisite = new java.util.Date();
        // le chemin des packages est nécessaire à cause de l'ambiguité sur le nom Date
        String dateDerVisiteTexte = DateFormat.
            getDateTimeInstance(DateFormat.FULL, DateFormat.FULL, Locale.FRANCE).
            format(dateDerVisite);
        String dateDerVisiteTextePourCookie = dateDerVisiteTexte.replace(' ', '.') + "\n";
        // pour Internet Explorer
        Cookie cookieDerVisite = new Cookie ("derVisite", dateDerVisiteTextePourCookie);
        resp.addCookie(cookieDerVisite);

        // recherche des cookies éventuels
        String dateDerVisiteDansCookie=null;
        Cookie[] tabCookies = req.getCookies();
        if (tabCookies != null)
            for (int i=0; i<tabCookies.length; i++)
            {
                if ("derVisite".equals(tabCookies[i].getName()))
                    dateDerVisiteDansCookie = tabCookies[i].getValue();
            }

        resp.setContentType("text/text");
        PrintWriter sortie = resp.getWriter();
        String nomClient, prenomClient, sexeClient;
        nomClient = req.getParameter("nom");
        prenomClient = req.getParameter("prenom");
        sexeClient = req.getParameter("sexe");

        boolean existeDeja = false;
        int nbreClients=0;
    }
}

```

```

try
{
    Class.forName("org.gjt.mm.mysql.Driver");
}
catch (ClassNotFoundException e)
{ sortie.println("Driver JDBC-OBDC non chargé " + e.getMessage()); }
try
{
    con = DriverManager.getConnection("jdbc:mysql://localhost/vilvens",
        "vilvens","xxxxx");
    sortie.println("Connexion à la BDD vilvens réalisée");
    instruc = con.createStatement();
    SyncSqlSelect("select * from clients");

    do
    {
        String n = rs.getString("nom"); String p = rs.getString("prenom");
        synchronized (this)
        {
            if (n.equals(nomClient) && p.equals(prenomClient)) existeDeja = true;
            else nbreClients++;
            sortie.println(nbreClients + ". " + " : " + n + " " + p);
        }
    }
    while (rs.next())

    sortie.println("Servlet en action !!!");
    if (dateDerVisiteDansCookie!=null)
    {
        sortie.println("Date de la dernière visite : ");
        sortie.println(dateDerVisiteDansCookie);
    }
    else
        sortie.println("C'est votre première visite");

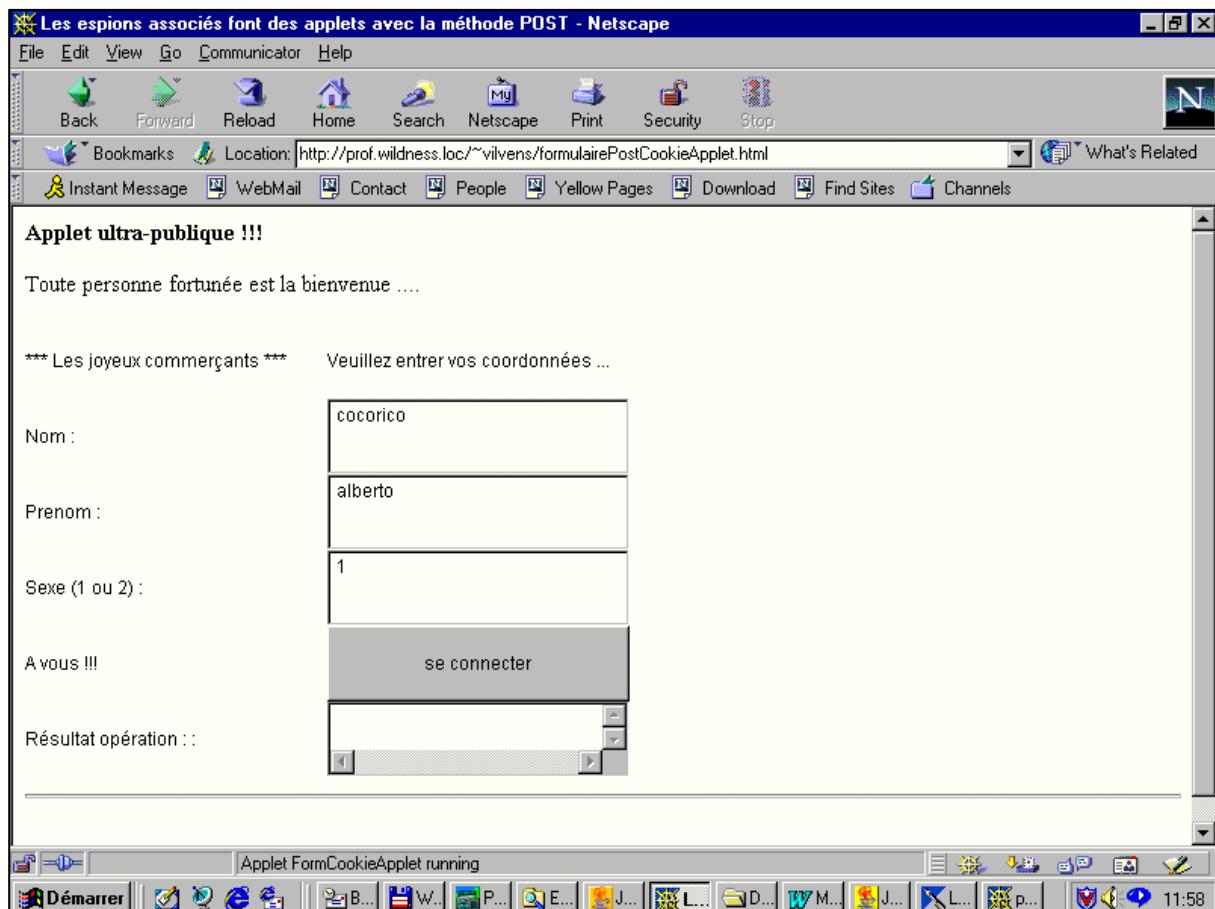
    sortie.println("Il y a : " + nbreClients + " clients");
    if (!existeDeja)
    {
        int numNouv = nbreClients+1;
        String numNouvClient = String.valueOf(numNouv);
        String numClient;
        if (sexeClient.equals("1"))numClient = new String("gen"+numNouvClient);
        else numClient = new String("gem"+numNouvClient);
        sortie.println("Numéro de client généré = " + numClient);
        int soldeParDefaut = 100;
        String requete = "insert into clients (numClient, nom, prenom, solde) values (" +
            + "" + numClient + "", "
            + "" + nomClient + "", "
            + "" + prenomClient + "", "
            + "" + soldeParDefaut + ")";
    }
}

```

```
        if (SyncSqlInsert(requete)) sortie.println("Problème d'insertion du tuple");
        else sortie.println("Insertion du tuple");
    }
    else sortie.println("Vous êtes déjà enregistré dans la base de données !!!");
}
catch (SQLException e)
{ sortie.println("Erreur JDBC-OBDC : " + e.getMessage()); }
sortie.close();
}

private synchronized boolean SyncSqlSelect (String requete) throws SQLException
{
    rs = instruc.executeQuery(requete);
    if (rs.next()) return true;
    else return false;
}
private synchronized boolean SyncSqlInsert (String requete) throws SQLException
{
    return instruc.execute(requete);
}
}
```

L'applet (FormCookieApplet) qui dialogue avec cette servlet est tout à fait analogue à celle développée dans les paragraphes précédents.



On obtient dans la zone résultat :

Connexion à la BDD vilvens réalisée
1. : vilvens claude
2. : mercenier denys
3. : clermont carine
4. : labelle francesca
5. : charlet christophe
Servlet en action !!!

C'est votre première visite

Il y a : 5 clients
Numéro de client généré = gen6
Insertion du tuple

Un 2^{ème} appui sur "se connecter" donne :

Connexion à la BDD vilvens réalisée
1. : vilvens claude
...
6. : cocorico alberto
Servlet en action !!!

Date de la dernière visite :
lundi.2.juillet.2001.12.h.05.GMT+02:00

Il y a : 6 clients
Vous etes déjà enregistré dans la base de données !!!

17.5 Un cookie pour l'identification d'une session

Il s'agit ici d'un autre exemple d'utilisation de cookie, particulièrement prisé dans le contexte de l'e-commerce. La servlet, lorsqu'elle est contactée, cherche **le cookie identifiant le client ou plutôt sa session** (ci-dessous, il s'appelle *idSession*). Si ce cookie est introuvable, la servlet le crée (l'identificateur est ici créé par un générateur aléatoire, mais on pourrait user d'un générateur RMI qui fournit un identificateur garanti unique ;-)).

FormCookieServlet.java (version idSession)

```
import javax.servlet.*;  
import javax.servlet.http.*;  
import java.io.*;  
import java.sql.*;  
import java.util.*;  
import java.text.*;  
  
public class FormCookieServlet extends HttpServlet  
{  
    Connection con;  
    Statement instruc;  
    ResultSet rs;
```

```

public void doPost(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, IOException
{
    // recherche des cookies éventuels
    String idDansCookie = null;
    Cookie[] tabCookies = req.getCookies();
    if (tabCookies != null)
        for (int i=0; i<tabCookies.length; i++)
    {
        if ("idSession".equals(tabCookies[i].getName()))
            idDansCookie = tabCookies[i].getValue();
    }

    String idNewDansCookie=null;
    boolean idExiste;
    if (idDansCookie == null)
    {
        idNewDansCookie = (new Integer((int)(Math.random()*1000))).toString() + "\n";
        Cookie cookieId = new Cookie ("idSession", idNewDansCookie);
        resp.addCookie(cookieId);
        idExiste=false;
    }
    else idExiste=true;

    resp.setContentType("text/text");
    PrintWriter sortie = resp.getWriter();
    if (!idExiste)sortie.println("Servlet en action !!! Vous recevez l'identifiant " +
        idNewDansCookie);
    else sortie.println("Servlet en action !!! Vous avez été reconnu pour " +
        idDansCookie);

    String nomClient, prenomClient, sexeClient;
    nomClient = req.getParameter("nom");
    prenomClient = req.getParameter("prenom");
    sexeClient = req.getParameter("sexe");

    boolean existeDeja = false;
    int nbreClients=0;
    try
    {
        Class.forName("org.gjt.mm.mysql.Driver");//.newInstance();
    }
    catch (ClassNotFoundException e)
    {
        sortie.println("Driver JDBC-OBDC non chargé " + e.getMessage());
    }
    try
    {
        con =
        DriverManager.getConnection("jdbc:mysql://localhost/vilvens","vilvens","xxxxxx");
    }
}

```

```

sortie.println("Connexion à la BDD vilvens réalisée");
instruc = con.createStatement();
SyncSqlSelect("select * from clients");
do
{
    synchronized(this) { nbreClients++; }
    String n = rs.getString("nom"); String p = rs.getString("prenom");
    if (n.equals(nomClient) && p.equals(prenomClient)) existeDeja = true;
    sortie.println(nbreClients + ". " + " : " + n + " " + p);
}
while (rs.next())

sortie.println("Servlet en action !!!");
sortie.println("Il y a : " + nbreClients + " clients");

if (!existeDeja)
{
    synchronized(this)
    {
        int numNouv = nbreClients+1;
        String numNouvClient = String.valueOf(numNouv);
        String numClient;
        if (sexeClient.equals("1"))numClient = new String("gen"+numNouvClient);
        else numClient = new String("gem"+numNouvClient);
        sortie.println("Numéro de client généré = " + numClient);
        int soldeParDefaut = 100;
        String requete = "insert into clients (numClient, nom, prenom, solde) values (" +
                        + """" + numClient + """",
                        + """" + nomClient + """",
                        + """" + prenomClient + """",
                        + """" + soldeParDefaut + ")");
        if (SyncSqlInsert(requete)) sortie.println("Problème d'insertion du tuple");
        else sortie.println("Insertion du tuple");
    }
}
else sortie.println("Vous etes déjà enregistré dans la base de données !!!");
}
catch (SQLException e)
{
    sortie.println("Erreur JDBC : " + e.getMessage());
    sortie.close();
}

private synchronized boolean SyncSqlSelect (String requete) throws SQLException {...}
private synchronized boolean SyncSqlInsert (String requete) throws SQLException { ... }
}

```

Evidemment, le procédé tombe à plat si le browser a été configuré pour refuser les cookies. Et puisque nous en sommes au suivi de session, figurez-vous que l'on trouve cela tout fait ;-) ...

Remarque

Une bonne idée serait de placer la connexion à la base de données dans la méthode init(), histoire d'éviter des recherches répétées du driver JDBC. Evidemment, des variables membres (instanciant Connection, Statement, Resultset ...) seront nécessaires.

18. Le suivi de session

18.1 L'interface HttpSession

On l'a dit et répété, HTTP est un protocole sans état, ce qui signifie qu'il ne conserve rien entre deux dialogues requête/réponse. Il est donc impossible à un serveur HTTP de reconnaître que des requêtes proviennent du même client, ce qui peut se révéler gênant pour un certain nombre d'applications. Un exemple classique est le "caddie virtuel" : un client, au moyen d'une page HTML et/ou d'une applet fait ses emplettes par Internet. Il est clair qu'il faut absolument reconnaître l'acheteur à chacun de ses achats ...

On peut contourner le problème en utilisant des *cookies* (voir ci-dessus) ou des *réécritures d'URL*. Cette dernière technique a pour principe de compléter l'URL de la servlet d'une information complémentaire (par exemple, un identificateur de session) – cette information se trouve entre le path et la query-string. Elle est récupérable au moyen de la méthode de HttpServletRequest :

```
public abstract String getPathInfo()
```

Si on ne récupère rien, c'est qu'il faut créer un nouveau numéro de session; sinon, on se contente de le récupérer.

Cependant, plus simplement et fort heureusement, le JSDK 2.0 fournit un "**API Session Tracking**" qui permet de gérer une "**session HTTP**" au moyen d'un interface **HttpSession**, figurant dans le package javax.servlet.http. L'implémentation du suivi de session varie selon les serveurs. Celle de base utilise les cookies. Mais certains serveurs sont capables de passer à la réécriture d'URL si la technique des cookies échoue : cette technique consiste à ajouter, entre l'URL proprement dite et la chaîne des paramètres du formulaire commençant par "?", des informations de session débutant par ";". Par exemple :

<http://eclipse.beauvilvens.org/EditProjet.do;jsessionid=E4BEC54AC29C863C590135332A10F97D?action=affiche&annee=2006-2007&projectId=121>

Evidemment, si HttpSession n'est qu'un interface, d'où proviendra l'objet qui l'implémente ? De l'objet HttpServletRequest qui sert de paramètre aux méthodes doGet() et doPost(); en effet, on y trouve la méthode

```
public abstract HttpSession getSession(boolean create)
```

qui, une fois implémentée par un véritable objet, fournit la session courante ou, si celle-ci n'existe pas et si le paramètre est à true, crée une nouvelle session. La méthode renvoie null si la session n'existe pas. Tout commencera donc par :

```
public void doGet(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, IOException
{
    ...
    HttpSession session = req.getSession(true);
    ...
}
```

Le rôle de l'objet session est clairement de conserver des informations concernant l'interaction entre le serveur et le client (plus exactement le navigateur de ce dernier). Ces informations peuvent être placées dans l'objet session par les méthodes de HttpSession :

```
public void putValue(String name, Object value)           [deprecated]
public void setAttribute(String name, Object value)
```

qui permettent d'associer un nom à une donnée au sein de la session. Bien sûr, on la récupérera au moyen de :

```
public abstract Object getValue(String name)           [deprecated]
public Object getAttribute(String name)
```

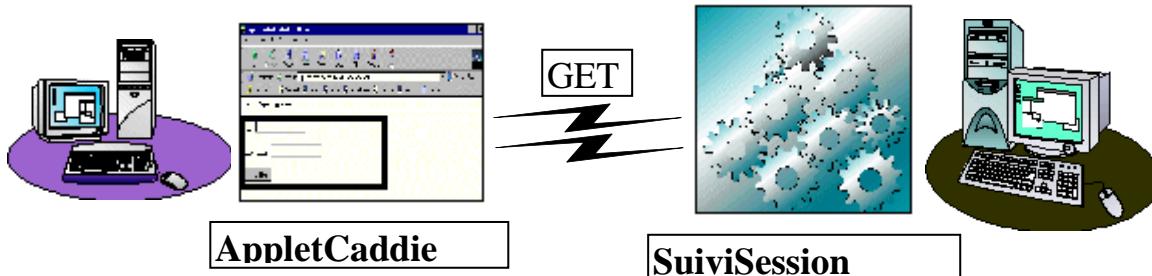
On obtient une référence nulle s'il n'existe pas de donnée correspondant au nom spécifié. De plus, on peut obtenir la liste des informations disponibles au sein de la session avec

```
public abstract String[] getValueNames()           [deprecated]
public java.util.Enumeration getAttributeNames()
```

A remarquer que ces méthodes lancent l'exception (non "checkée") IllegalStateException si la session est devenue invalide (par **invalidate()** – voir plus loin).

18.2 Les classiques : l'id de session et le compteur d'accès

Pour illustrer le suivi de session, l'exemple classique est d'enregistrer un "id de session" et aussi compter le nombre d'accès ("hits") réalisé par un client à partir de son browser. Nous imaginerons que celui-ci se promène dans un magasin virtuel pour remplir un caddie tout aussi virtuel; la page HTML, avec son applet, lui permet de désigner un article qu'il désire acheter, avec la quantité.



Visuellement, cela donnerait :

The screenshot shows a Java applet window titled "Le Grand Magasin !!!!". Inside, there's a message: "Faites vos achats en toute tranquilité depuis votre fauteuil !" followed by "*** La joie de dépenser ***". Below this, there are two input fields: "Article:" containing "robe "Venus" bleue" and "Quantité:" containing "1". At the bottom left is a button labeled "N'hésitez plus !", and at the bottom right is a blue button labeled "J'achète :-)".

L'applet correspondante (générée avec NetBeans) s'écrit sans surprise :

SuiviSessionAppletAchat.java

```
package SessionTracking;

import java.applet.*;
import java.net.*;

/**
 * @author Vilvens
 */
public class SuiviSessionAppletAchat extends javax.swing.JApplet
{
    public void init(){ ... }
    private void initComponents() { ... }

    private void BConnecterActionPerformed(java.awt.event.ActionEvent evt)
    {
        String a = article.getText(); String q = quantite.getText();
        String adresseServlet = "/SuiviSessionServlet/ServletAchat"
            + "?article=" + URLEncoder.encode(a) + "&quantite=" + URLEncoder.encode(q);
        URL urlServlet = null;
        try
        {
            URL pageCourante = getDocumentBase();
            System.out.println("getDocu = " + pageCourante);
            String protocole = pageCourante.getProtocol();
            String machine = pageCourante.getHost();
            int port = pageCourante.getPort();
            System.out.println("p=" + protocole + " m=" + machine + " port=" + port);
            urlServlet = new URL(protocole, machine, port, adresseServlet);
        }
    }
}
```

```

        System.out.println("urlServlet =" + urlServlet.toString());
        getServletContext().showDocument(urlServlet);
    }
    catch (MalformedURLException ex)
    {
        System.out.println("Aie aie - URL louche ! : " + ex.getMessage());
    }
}

private javax.swing.JButton BConnecter;
private javax.swing.JTextField article;
...
private javax.swing.JTextField quantite;
}

```

La servlet (nommons-la *SuiviSessionServlet*), pour sa part,

- ◆ lit à chaque soumission l'article et la quantité entrés par le client
- ◆ recherche un identifiant de session (appelons-le "session.identifiant") dans l'objet session obtenu à partir du paramètre requête la méthode doGet()/doPost() (donc processRequest() dans le cas d'une servlet générée avec NetBeans); elle en invente un si elle n'en trouve pas;
- ◆ se réfère à un compteur (appelons-le "session.compteur") qui est mémorisé dans l'objet session; la servlet numérote les achats et devrait, normalement, les enregistrer dans une base de données (nous nous dispenserons de cette étape).

Cela donne :

SuiviSessionServlet.java

package *SessionTracking*;

```

import java.io.*;
import java.net.*;

import javax.servlet.*;
import javax.servlet.http.*;

/**
 * @author Vilvens
 */
public class SuiviSessionServlet extends HttpServlet
{
    protected void processRequest(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException
    {
        resp.setContentType("text/html;charset=UTF-8");
        PrintWriter sortie = resp.getWriter();
        sortie.println("<HTML><HEAD><TITLE>");
        sortie.println("Réponse de la servlet aux requêtes client");
        sortie.println("</TITLE></HEAD><BODY>");
    }
}

```

```

sortie.println("<H3>Le Grand Magasin !!!!!</H3><p>");
sortie.println("<p>Achat de " + req.getParameter("quantite") + " "
+ req.getParameter("article") + " !!!<p>");

HttpSession session = req.getSession(true);

String idSession = (String)session.getAttribute("session.identificateur");
if (idSession==null)
{
    idSession = (new Integer((int)(Math.random()*1000))).toString() + "\n";
    session.setAttribute("session.identificateur", idSession);
    sortie.println("<P><P>Numéro de session attribué : " + idSession + "<p>");
}
else sortie.println("<P><P>Numéro de session récupéré : " + idSession + "<p>");

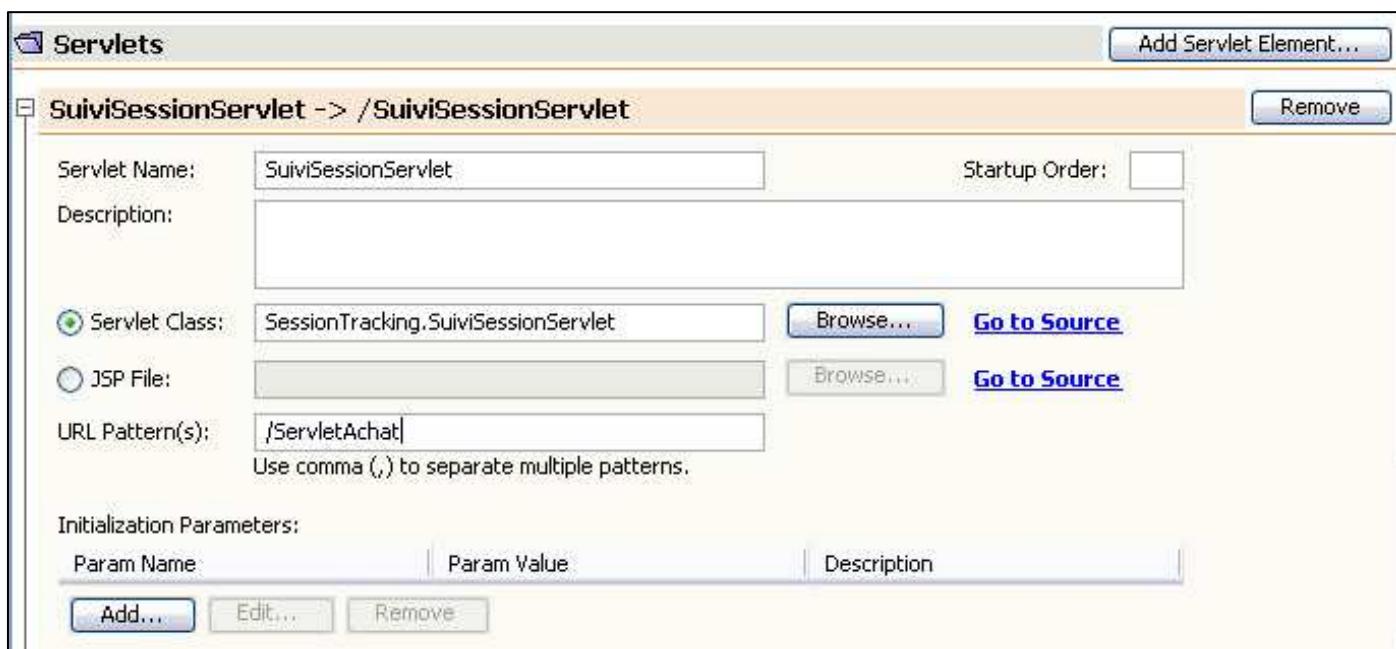
Integer nbreRequetes = (Integer) session.getAttribute("session.compteur");
if (nbreRequetes==null) nbreRequetes = new Integer(1);
else nbreRequetes = new Integer (nbreRequetes.intValue()+1);
session.setAttribute("session.compteur", nbreRequetes);

sortie.println("<P><P>Achat numéro " + nbreRequetes + " enregistre ...<p>");
sortie.println("<p>Méthode utilisée = " + req.getMethod() + "<p>");
sortie.println("<p>Protocole utilisé pour l'envoi de la requête = " +req.getScheme() + "<p>");
sortie.println("<p>Host = " + req.getServerName() + "<p>");
sortie.println("</BODY></HTML>");
sortie.close();
}

protected void doGet(...) { ... }
protected void doPost(...) { ... }
public String getServletInfo(){ ... }
}

```

L'URL Pattern de la servlet a été modifié :

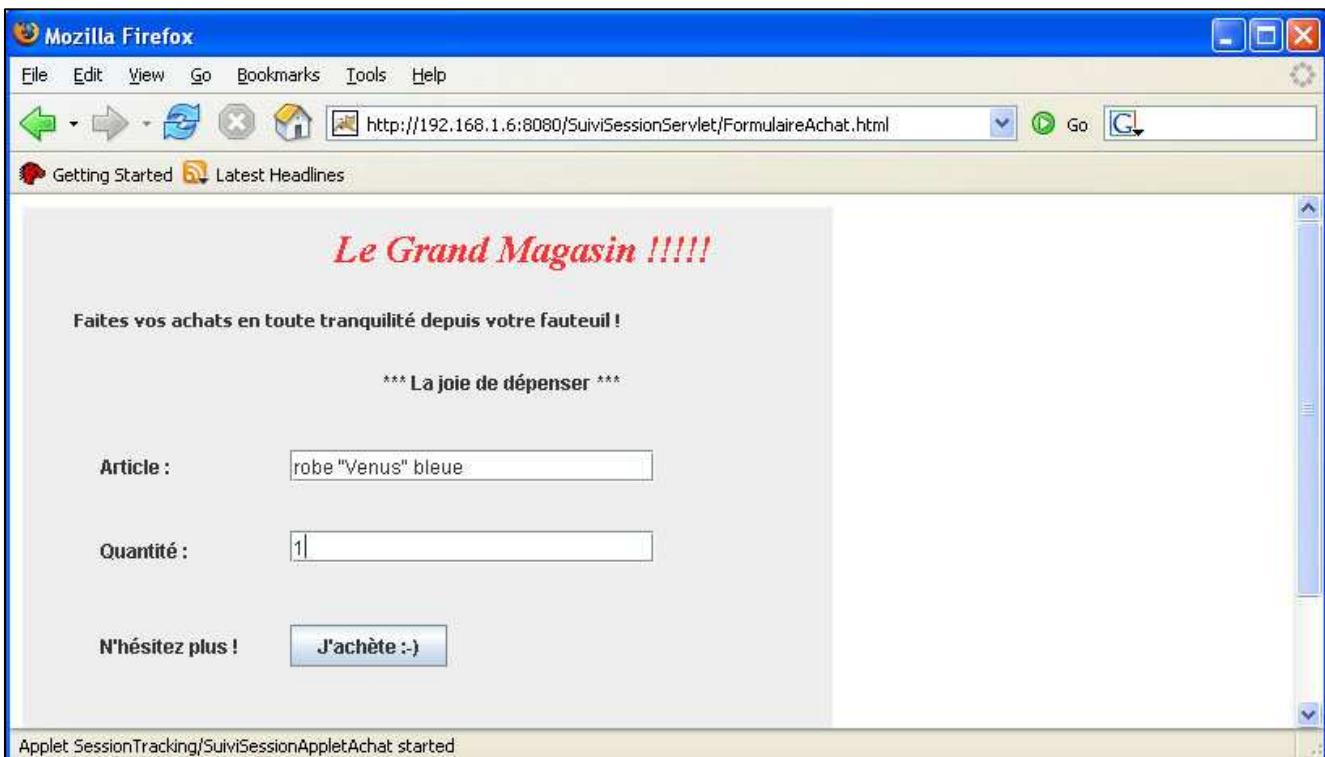


Tout démarre avec la page HTML :

FormulaireAchat.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head><title></title></head>
<body>
<applet code="SessionTracking/SuiviSessionAppletAchat.class"
archive="SuiviSessionApplet.jar" width=500 height=400></applet>
</body>
</html>
```

Une chaîne d'exécutions pourrait donner :





18.3 Le cycle de vie d'une session

Le cycle de vie d'une session n'est pas aussi clair que celui d'une applet ou d'une servlet. En fait, une session peut expirer :

- ◆ soit automatiquement après un certain temps d'inactivité, dont la valeur est fixée par le serveur;
- ◆ soit explicitement, par l'utilisation de la méthode

`public abstract void invalidate()`

Par l'autre bout de la lorgnette, une session est considérée comme nouvelle si elle a été créée par le serveur mais que le client n'a pas encore créé le lien avec elle. La méthode

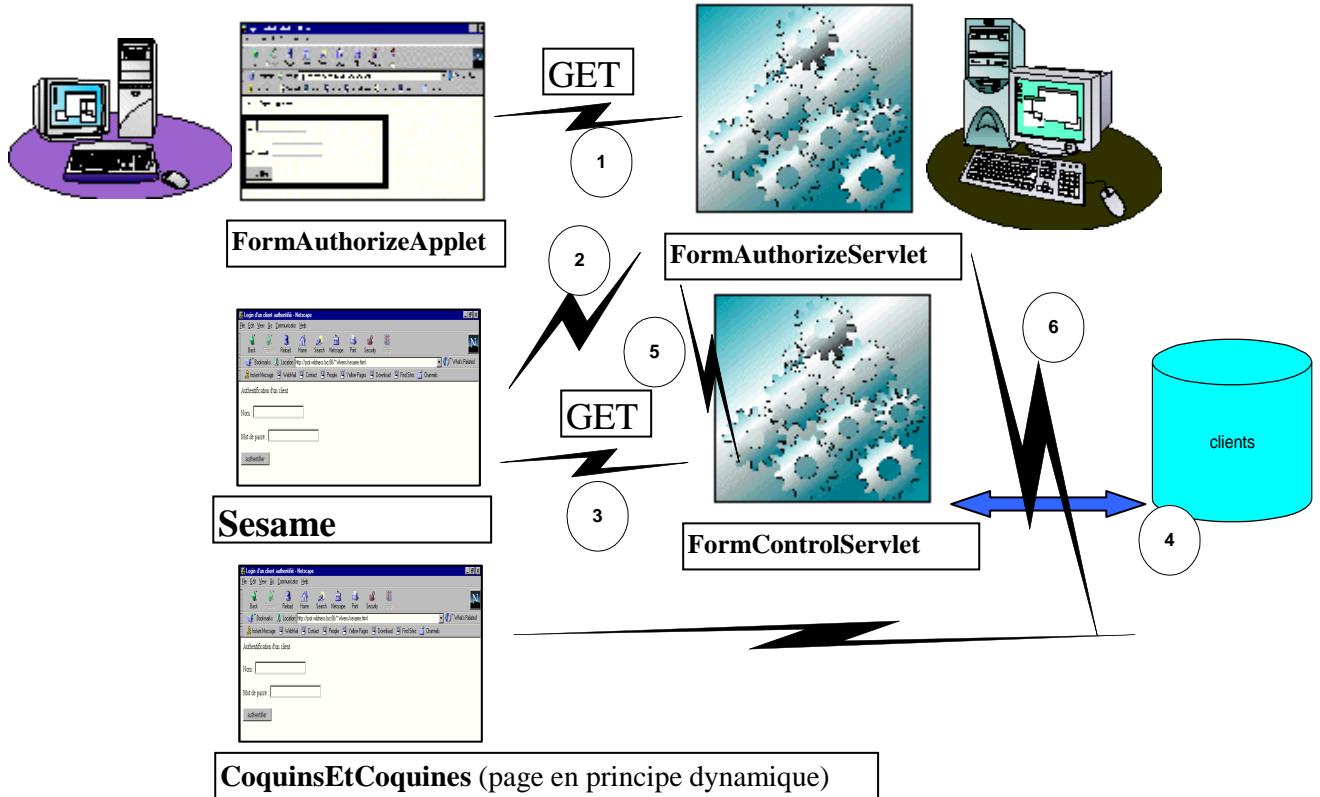
public abstract boolean isNew()

permet de tester cela.

19. L'authentification d'un client

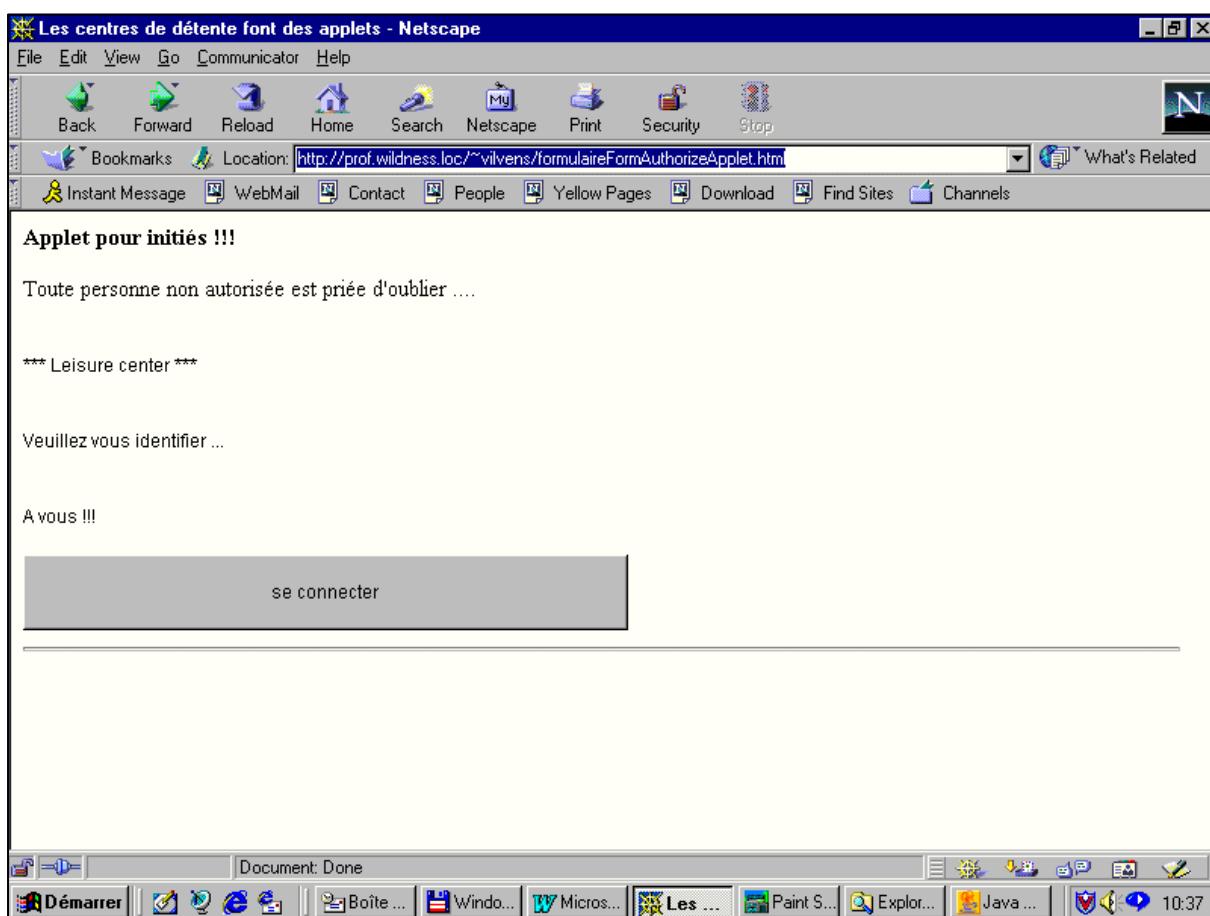
19.1 Les acteurs

Nous allons à présent utiliser le suivi de session dans un autre scénario classique d'utilisation, celui d'une applet avec deux servlets, l'une d'accès et l'autre d'authentification.



19.2 L'applet de connexion

Un client (au sens commercial du terme) veut se connecter à un site protégé. Il le fait, par exemple, au moyen d'une applet fonctionnant au sein d'une page HTML :



L'applet ("from scratch") s'écrira simplement :

FormAuthorizeApplet.java

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.net.*;

public class FormAuthorizeApplet extends Applet implements ActionListener
{
    TextField nom, prenom;
    public void init()
    {
        resize(400,200);
        setLayout(new GridLayout(4,2));
        add(new Label("*** Leisure center ***"));
        add(new Label("Veuillez vous identifier ..."));
        add(new Label("A vous !!!"));
        Button seConnecter = new Button ("se connecter"); add(seConnecter);
        seConnecter.addActionListener(this);
    }
}
```

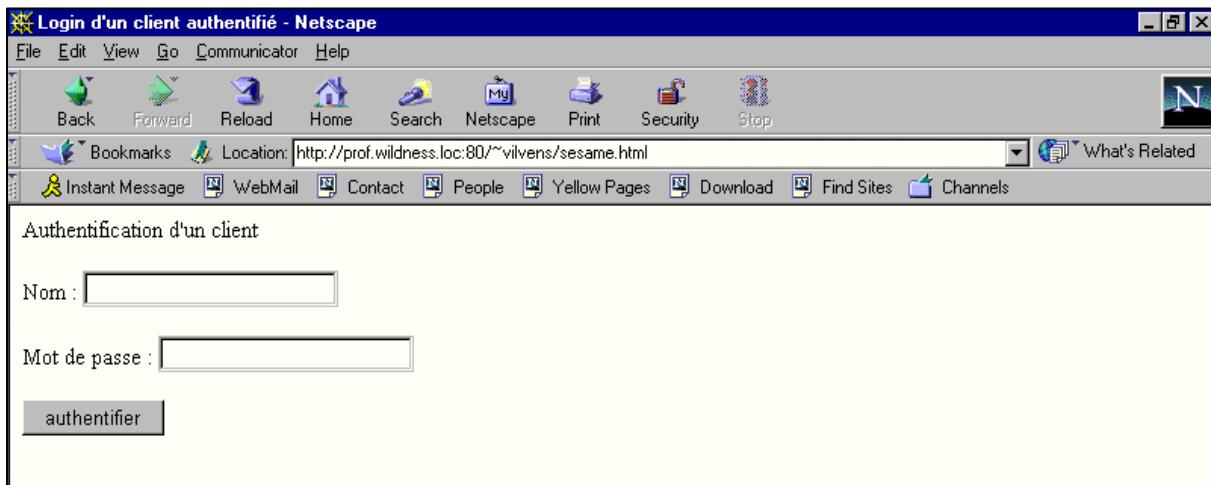
```

public void actionPerformed(ActionEvent e)
{
    if (e.getActionCommand() == "se connecter")
    {
        String adresseServlet = "/vilvens/servlet/FormAuthorizeServlet";
        URL urlServlet = null;
        try
        {
            URL pageCourante = getDocumentBase();
            System.out.println("getDocu = " + pageCourante);
            String protocole = pageCourante.getProtocol();
            String machine = pageCourante.getHost();
            int port = pageCourante.getPort();
            System.out.println("p=" + protocole + " m=" + machine + " port=" + port);
            urlServlet = new URL(protocole, machine, port, adresseServlet);
        }
        catch (MalformedURLException ex)
        {
            System.out.println("Aie aie - URL louche ! : " + ex.getMessage());
        }
        getAppletContext().showDocument(urlServlet);
    }
}
}

```

19.3 La servlet d'accès

Sa tentative l'emmènera à un page de "login" qui lui permettra d'introduire son nom et son mot de passe.



La servlet associée à ce travail vérifiera si ce client est connu. Si oui, elle lui donnera l'accès à ce qu'il demandait. Mais comment "se souvenir" de ce que le client demandait ? Par le suivi de session ... Un objet HttpSession permettra de savoir si le client s'est déjà authentifié : l'information "*logon.isDone*" servira à cela. Si elle n'existe pas, il est clair qu'il faudra détourner le client vers une page de login pour renvoyer ensuite vers la destination originale. La méthode de HttpServletResponse :

public abstract void sendRedirect(String location) throws IOException

permet précisément de fournir au client une réponse temporaire qui provient de la nouvelle URL spécifiée comme paramètre (cette URL doit être absolue) – autrement dit, on renvoie le client vers une autre page ou une autre servlet. Pour assurer le retour, on sauvegardera dans l'objet session une information "*login.target*" dépositaire de l'URL initialement demandée dans la requête. Celle-ci est obtenue au moyen de la méthode

```
public static StringBuffer getRequestURL(HttpServletRequest req)
```

de la classe utilitaire HttpUtils. Cela donne (avec les méthodes "deprecated" *putValue()* et *getValue()*) :

FormAuthorizeServlet.java

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class FormAuthorizeServlet extends HttpServlet
{
    int nbreConnexions;

    public void init (ServletConfig config) throws ServletException
    {
        super.init(config);
        nbreConnexions = 0;
    }

    public void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException
    {
        resp.setContentType("text/html");
        PrintWriter sortie = resp.getWriter();

        sortie.println("<HTML><HEAD><TITLE>");
        sortie.println("Réponse de la servlet à l'accès demandé");
        sortie.println("</TITLE></HEAD><BODY>");
        sortie.println("<H1>Authentification en cours</H1>");

        HttpSession session = req.getSession(true);
        Object existe = session.getValue("logon.isDone");
        if (existe==null)
        {
            session.putValue("login.target", HttpUtils.getRequestURL(req).toString());
            sortie.println(HttpUtils.getRequestURL(req).toString());
            resp.sendRedirect (req.getScheme()+"://"+req.getServerName()+
                ":"+req.getServerPort()+"/~vilvens/sesame.html");
            return;
        }

        sortie.println("<p><B>Authentification réussie !!!<b><p>\"");
        synchronized (this) { nbreConnexions++; }
```

```

sortie.println("<P><P>Vous etes notre client numero " +
    Integer.toString(nbreConnexions) + " ...<p>");
sortie.println("<p>Méthode utilisée = " + req.getMethod() + "<p>");
sortie.println("<p>Protocole utilisé pour l'envoi de la requete = " + req.getScheme() + "<p>");
sortie.println("<p>Host = " + req.getServerName() + "<p>");
sortie.println("Vous pouvez à présent parcourir notre site cochon ;-)! ");
sortie.println();
"<a href= \"http://prof.wildness.loc/~vilvens/coquinsEtCoquines.html\">coquins et coquines</a>"); // page dynamique : sinon, pourquoi ne pas l'accéder directement ;-)
sortie.println("</BODY></HTML>");
sortie.close();
}
}

```

La page de login est un simple formulaire :

sesame.html

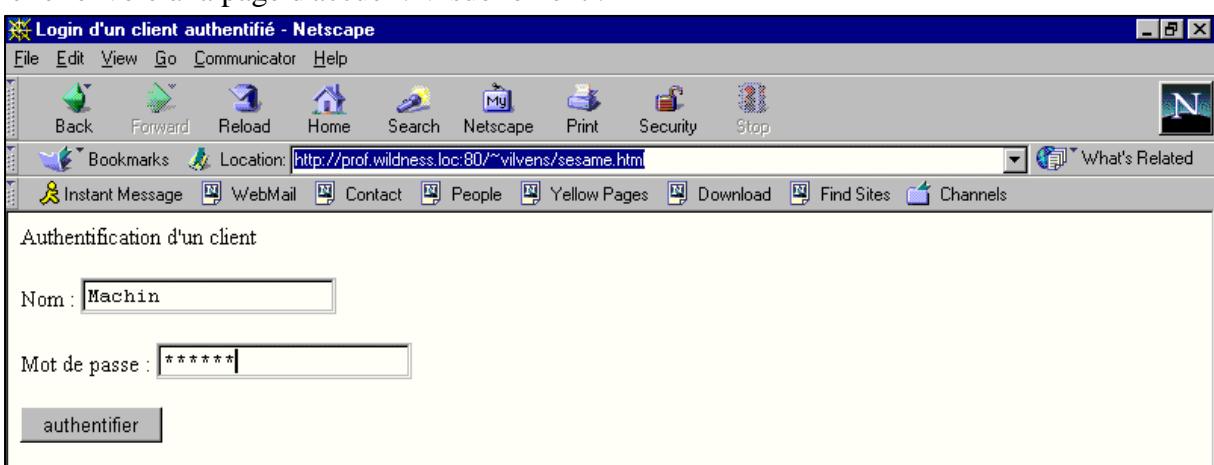
```

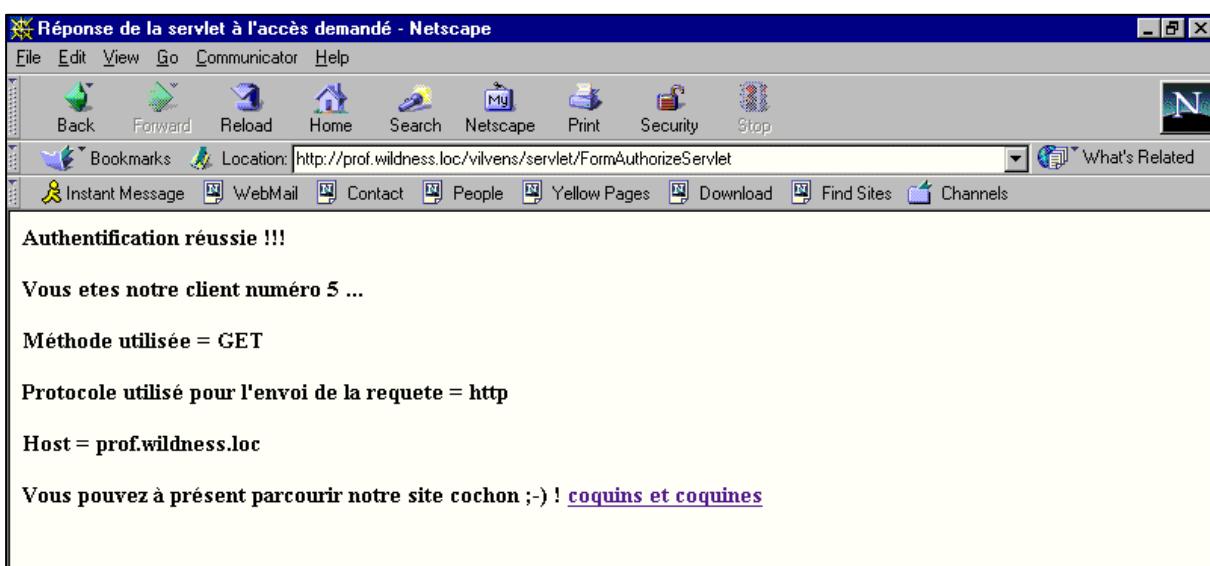
<!doctype html public "-//w3c//dtd html 4.0 transitional//en">
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
    <title>Login d'un client authentifi&eacute;</title>
</head>
<body>
Authentification d'un client&nbsp;
<form method="POST"
action="http://prof.wildness.loc/vilvens/servlet/FormControlServlet">
<p>Nom :&nbsp;<input type="text" name="nom" size=20>
<p>Mot de passe :&nbsp;<input type="password" name="motdepasse" size=20>
<p><input type="submit" value="authentifier">
<br></form>
</body>
</html>

```

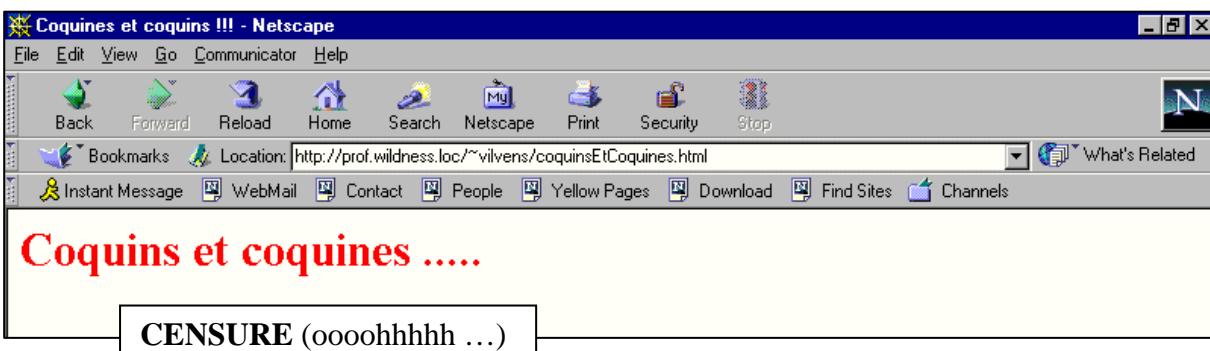
19.4 La servlet de contrôle

Cette servlet est contactée si l'information *logon.isDone* n'existe pas. Elle va alors authentifier, ou pas, le client connecté et renvoyer vers la servlet initiale ou, en cas d'erreur, elle renvoie à la page d'accueil. Visuellement :





Un petit clic sur le lien ...



La servlet de contrôle s'écrit comme suit :

FormControlServlet.java

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.sql.*;

public class FormControlServlet extends HttpServlet
{
    public void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException
    {
        resp.setContentType("text/html");
        PrintWriter sortie = resp.getWriter();

        // récupération des données du formulaire
        String nom = req.getParameter("nom");
        String motDePasse = req.getParameter("motdepasse");

        sortie.println("<HTML><HEAD><TITLE>");
        sortie.println("Réponse de la servlet à la demande d'authentification");
```

```

sortie.println("</TITLE></HEAD><BODY>");
sortie.println("<H1>Accueil d'un nouveau client</H1>");
sortie.println("Servlet en action !!!");

Connection con;
Statement instruc;
ResultSet rs;
int nbreClients=0;
try
{
    Class.forName("org.gjt.mm.mysql.Driver");
}
catch (ClassNotFoundException e)
{
    sortie.println("Driver JDBC-OBDC non chargé " + e.getMessage());
}
try
{
    con = DriverManager.getConnection("jdbc:mysql://localhost/vilvens",
        "vilvens","xxxxxx");
    sortie.println("Connexion à la BDD vilvens réalisée");
    instruc = con.createStatement();
    rs = instruc.executeQuery("select * from etcPassword");
    while (rs.next())
    {
        nbreClients++;
        String nom = rs.getString("nom"); String pwd = rs.getString("pwd");
        sortie.println(nbreClients + ". " + nom + " : " + pwd);
    }
    sortie.println("Il y a : " + nbreClients + " clients");
}
catch (SQLException e)
{
    sortie.println("Erreur JDBC-OBDC : " + e.getMessage());
}
sortie.println("</BODY></HTML>");

// recherche dans la base de données .....
// supposons que c'est OK
HttpSession session = req.getSession(true);
session.putValue("logon.isDone", nom);
try
{
    String target = (String)session.getValue("login.target");
    if (target!=null)
    {
        resp.sendRedirect(target);
        return;
    }
}

```

```

        catch (Exception e)
    {
        sortie.println("Erreur pour la redirection : " + e.getMessage());
    }
    // redirection impossible - on prend la page d'accueil
    resp.sendRedirect(req.getScheme() + "://" + req.getServerName() + ":" +
        req.getServerPort() + "/~vilvens/");
}
}

```

20. Le ServletContext

Nous avons vu que plusieurs servlets peuvent parfaitement être exécutées simultanément par le même moteur à servlets. On a alors coutume de dire qu'elles partagent le même **environnement** ou, mieux encore, **elles font partie de la même application WEB** puisqu'elles partagent le même web.xml, les mêmes répertoires classe et lib, etc. Il peut alors être intéressant d'obtenir, au sein de ces diverses servlets, des informations sur cet environnement, donc finalement sur les autres servlets – avec les risques et problèmes éventuels que cela peut comporter.

La spécification des servlets fournit un interface **ServletContext** (dans le package javax.servlet) dont le rôle est précisément de fournir l'accès à ces informations partagées au sein d'une application WEB. C'est au container WEB de fournir une implémentation de cet interface. Parmi ses méthodes, relevons en première analyse :

- ◆ public abstract Enumeration **getServletNames()**
fournit évidemment le nom des servlets actives sur le serveur
- ◆ public abstract Servlet **getServlet(String name)** throws ServletException
fournit l'objet servlet dont le nom est fourni (null si il n'existe pas)
- ◆ public abstract void **log(String msg)**
- ◆ public void **log(String message, Throwable throwable)**
cette méthode polymorphe permet de programmer des écritures dans un **fichier de journalisation**; on écrit soit un simple message, soit le message avec la pile des appels (car le paramètre Throwable possède une méthode printStackTrace()); le nom et l'emplacement du fichier log dépendent du serveur utilisé :

- pour NetBeans, c'est la fenêtre Output, onglet "Bundled Tomcat Log";
- pour Tomcat en standalone sous Windows, c'est un groupe de fichiers dont le nom est formé du composant de Tomcat concerné et de la date (par exemple, catalina.2006-10-31.log), fichiers placé dans le répertoire C:\Program Files\Apache Software Foundation\Tomcat 5.5\logs;
- pour Tomcat en standalone sous Linux, c'est dans le fichier /usr/tomcat4/jakarta-tomcat-4.1.30/log\$localhost_pw_log.<date du jour>.txt que cela se passe.

Evidemment, ServletContext n'est qu'un interface. D'où proviendra l'objet qui l'implémente ? De la méthode de GenericServlet :

```
public ServletContext getServletContext()
```

qui, en fait, appelle la méthode du même nom de l'objet implémentant l'interface **ServletConfig**, objet fourni à la servlet par le container Web dans la méthode init() de celle-ci..

En modifiant la servlet CptServletContext développée plus haut, on peut alors écrire :

CptServletContext.java

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;

public class CptServletContext extends HttpServlet
{
    private int compteur = 0;

    public void doGet (HttpServletRequest request, HttpServletResponse response)
        throws IOException
    {
        PrintWriter out;
        response.setContentType("text/html");
        out = response.getWriter();

        ServletContext sc = getServletContext();
        sc.log("-- demarrage de la servlet");

        out.println("<HTML><HEAD><TITLE>");out.println("Contexte des servlets");
        out.println("</TITLE></HEAD><BODY>");
        out.println("<H1>Examen du contexte</H1>");

        sc.log("-- incrementation du compteur");
        synchronized (this)
        {
            compteur++;
            out.println("Vous etes le client n° " + compteur + " !!!<P>");
        }

        out.println("Servlets chargées sur le serveur :<p><P>");
        int cpts = 0;

        sc.log("-- balayage des servlets");

        Enumeration nomsServlet = sc.getServletNames();
        while (nomsServlet.hasMoreElements())
        {
            cpts++;
            String nomServlet = ()nomsServlet.nextElement();

            Servlet s = null;
```

```

try
{
    s = sc.getServlet(nomServlet);
}
catch (ServletException e)
{
    out.println("??? Erreur : " + e.getMessage());
}
out.println(cpts + ". ** Nom de la servlet : " + nomServlet);

if (s!=null)
{
    out.println("*Classe de la servlet : " + s.getClass().getName());
    out.println("Infos sur la servlet : " + s.getServletInfo());
}
out.println("<br>");

out.println("</BODY></HTML>");
out.close();
}

public String getServletInfo()
{
    return "Servlet fournissant des informations sur l'environnement du moteur à servlets";
}
}

```

On remarquera, en passant :

- ◆ l'usage de la méthode d'Object⁷

public final native Class getClass()

qui retrouve un objet **Class** – nous avons rencontré cette classe dans le chapitre consacré à JDBC;

- ◆ la redéfinition de la méthode de GenericServlet :

public String getServletInfo()

a) Avec l'ancienne spécification des servlets, on peut invoquer différentes servlets et si elles ont été invoquées chacune au moins une fois, on obtiendra :

⁷ évoquée au chapitre II de "Langage Java (I) : Programmation de base" ...

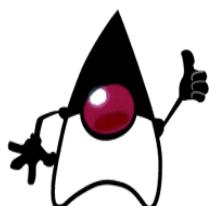


b) Malheureusement, **tout ceci ne fonctionne qu'avec l'ancienne spécification des servlets !** En effet, les méthodes `getServletNames()` et `getServlet()` sont à présent deprecated ! Et il n'y a pas de solution de remplacement : la référence d'Enumeration obtenue est simplement nulle. Dommage ...

Ceci dit, le `ServletContext` nous sera encore utile, notamment parce qu'il est capable de conserver des *informations partageables entre les membres d'une application Web*. En fait, le `ServletContext` dispose, *comme l'objet session*, des méthodes:

```
public void setAttribute(String name, Object value)  
public Object getAttribute(String name)
```

qui vont permettre de mémoriser ces informations partageables. Cette possibilité, avec d'autres, nous sera utile lorsque nous aurons fait plus ample connaissance avec les JSP ...



Ce fut long, mais alors vraiment long ! A chaque chapitre suffit (largement) sa peine. D'autant que les servlets possèdent des cousines : les Java Server Pages. Nous pouvons donc envisager de nous y attaquer ...

XVII. Les Java Server Pages



La vérité ne fait pas tant de bien dans le monde que ses apparences y font de mal.

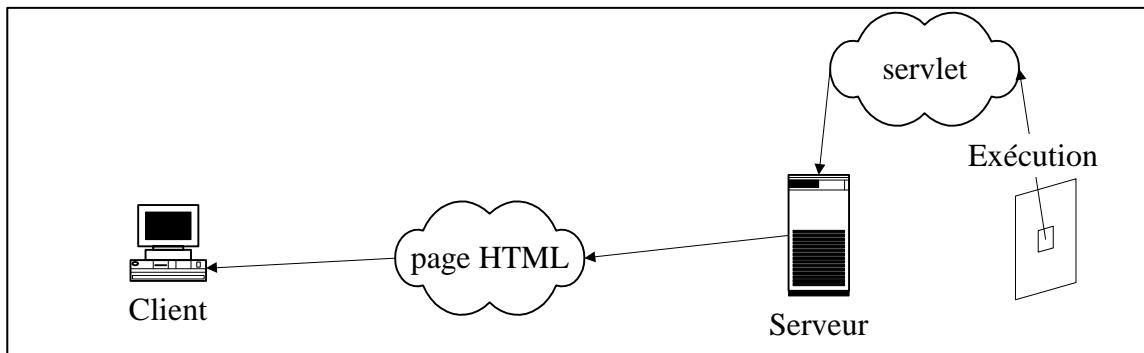
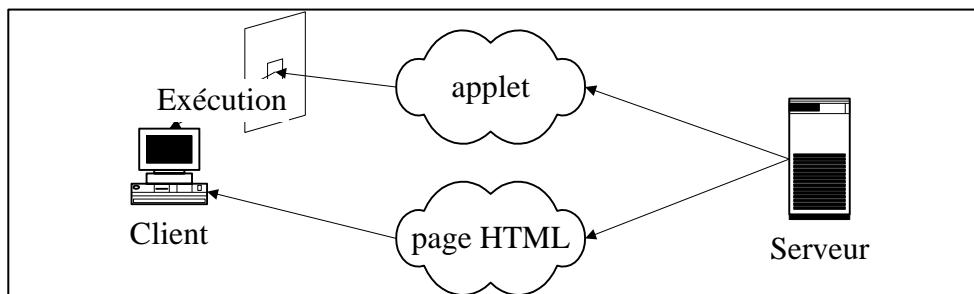
(F. de La Rochefoucauld, Maximes)

1. La technique SSI

Jusqu'à présent, nos servlets ont construit l'intégralité de la page dynamique que le client reçoit. Mais on pourrait aussi imaginer que seule une certaine partie de la page soit fabriquée dynamiquement, au sein d'une page HTML statique pour le reste. C'est ce que propose la technique **server-side includes (SSI)**. Dans ce contexte, une page HTML comporte en certains points des balises <SERVLET ...> et </SERVLET> qui permettent de désigner la servlet à activer et qui indiquent aussi le point d'insertion de la sortie de cette servlet. Concrètement, cette "Servlet Tag Technology" s'implémente ainsi :

```
<HTML> ...
<BODY> ...
    <SERVLET code="maServlet.class">
        <param nom="vil" prenom="clodius" />
    </SERVLET>      ...
</BODY>
/SERVLET>
```

Les tags sont donc similaires aux balises <APPLET ...> et </APPLET>. Attention cependant à cette analogie : ils sont bel et bien décodés par le serveur, et pas par le browser du client.



2. Les JSP

Dans le même esprit, les **JSP** (Java Server Pages) permettent de créer dynamiquement une partie de page HTML stockée sur le serveur, le reste de la page étant statique. La création de la zone dynamique est programmée en Java au sein de la page HTML et est effectivement réalisée du côté du serveur : c'est évidemment la grande différence avec un langage de script comme JavaScript qui crée également une partie de page HTML, mais du côté du client. Le fonctionnement des JSP rappelle donc aussi celui de PHP, si ce n'est que c'est le langage Java lui-même qui est utilisé, avec tout ce que cela signifie en termes de portabilité, de sécurité, d'utilisation potentielle de JavaBeans, etc.

La démarche des JSP n'est évidemment pas sans rappeler celle des servlets (et pour cause, comme nous allons le voir ...). En particulier, le modèle des JSP est multithread : plusieurs threads peuvent avoir accès simultanément à une page JSP (sauf si on spécifie explicitement le contraire).

La spécification actuelle des JSP est 2.0 (la précédente était 1.2, parfaitement compatible avec 2.0). Un serveur Web actuel se doit d'être capable de les gérer. C'est le cas, par exemple, d'Apache dans lequel on a "plugué" le moteur à servlets/JSP **Tomcat 5/6** ou de Tomcat lui-même utilisé en standalone.

3. Un exemple introductif

Avant de nous livrer à une analyse plus poussée, nous allons immédiatement tenter de donner une petite idée de l'utilisation des JSP. Supposons donc vouloir obtenir une page Web qui affiche "Bonjour !" un certain nombre de fois (intellectuel, non ;-)?). Le nombre d'itérations sera au préalable fixé par l'utilisateur, qui choisira ce nombre sur l'URL d'appel depuis le browser ou, de manière plus agréable, au moyen d'un formulaire.

Le fichier RepeteBonjour.jsp qui réalisera la tâche annoncée ressemble à ceci :

RepeTeBonJour.jsp

```
<%@ page language="java" %>
<%@page contentType="text/html"%>
<HTML>
<HEAD>
<TITLE>Bonjour à répétition</TITLE>
<!-- Premier essai de JSP -->
<%-- C'est Vilvens qui m'a faite --%>
</HEAD>
<BODY>
*** Accueil dans le monde des JSP ***
<p>

<%    int nbre = Integer.parseInt(request.getParameter("nombre"));
      for (int i=0; i<nbre; i++)
      {
%
      >        Bonjour ! <BR>
<%    }
%
      ></P>
Cela nous fait <%=nbre %> fois ...
</BODY>
</HTML>
```

En première approximation, il s'agit donc bien d'une page HTML, mais avec des zones qui seront construites au moment de l'envoi vers le client. Ces zones sont encore appelées

- ◆ des **scriptlets** si elles contiennent du code Java; elles sont déterminées par les tags `<%>`; on aura compris que ces scriptlets sont destinées à contenir le code Java (y compris des déclarations de variable ou des définitions de méthodes); on peut remarquer ci-dessus qu'une scriptlet peut laisser ouverts un ou plusieurs blocs d'instruction qui seront refermés ultérieurement par une autre scriptlet;
- ◆ des **directives** si elles définissent des attributs de la future page, attributs qui resteront constants pour toutes les requêtes; elles sont déterminées par les tags `<%@>`; ainsi,
 - le fait que le code des scriptlets est du Java est indiqué par la directive de page `<%@ page language="java" %>` qui indique le langage support;
 - l'autre directive `<%@page contentType="text/html" %>` précise évidemment le type MIME de la réponse qui va être générée et envoyée par le JSP.
- ◆ des **expressions** dont le résultat de l'évaluation sera placé dans la page à la place de l'expression; elles sont déterminées par les tags `<%= %>`; ici, c'est simplement l'évaluation de la variable `nbre` qui sera placée (mais il pourrait aussi s'agir de l'appel d'une méthode); on remarquera l'absence de point-virgule à la fin de l'expression;
- ◆ des **commentaires** si elles sont bordées de `<%-- --%>`; ils sont bien entendu ignorés.

On peut encore remarquer que le texte affiché sur la page ("Bonjour !
") ne fait pas partie du code Java !

On peut signaler qu'il existe encore d'autres types de zones, mais qui n'apparaissent pas ici :

- ◆ les **déclarations de variables ou de méthodes** (membres de la future servlet résultant de la compilation de la page JSP – mais n'anticipons pas), déterminées par les tags `<%!>` et `<%>`; par exemple :

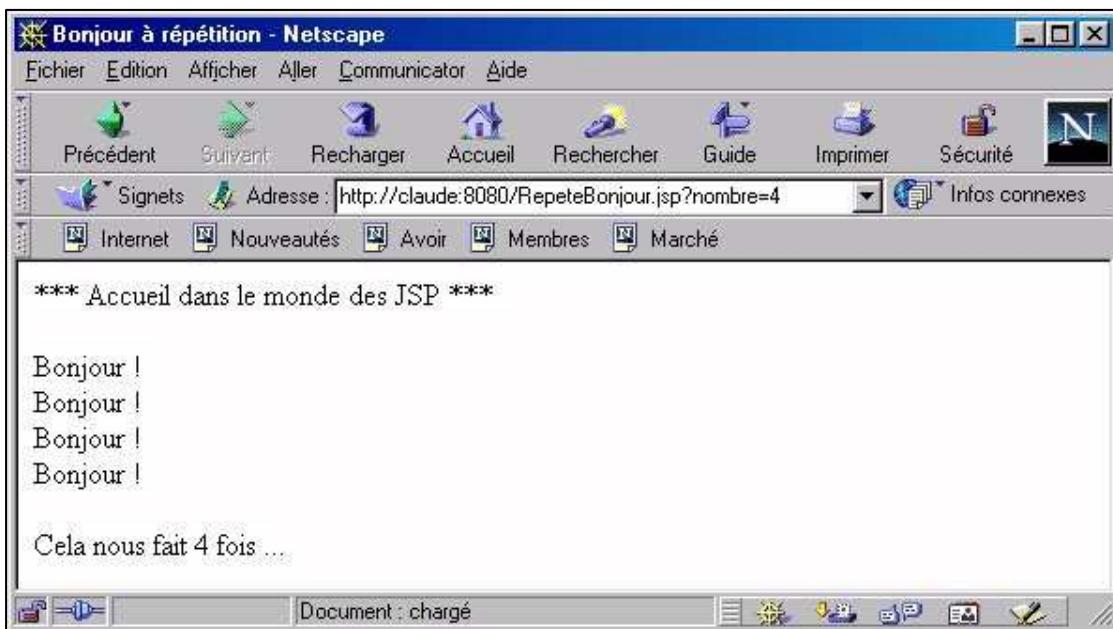
```
<%! int cpt=0; %>
<%! public int Fois2 (int x)
{
    returnx*2;
} %>
```

- ◆ les **actions**, indiquées par des tags spécifiques comme `<%jsp:...>`, pour gérer par exemple les beans et les plugins.

Lors de l'appel du JSP, par exemple dans la zone URL d'un browser :

<http://claude:8080/RepeTeBonjour.jsp?nombre=4>

le code JSP va être compilé par le moteur à servlets (plus précisément son compilateur de pages) **en une servlet** et c'est cette servlet qui sera, en définitive, compilée puis exécutée : la page résultante sera envoyée au client, comme d'habitude, comme une page statique normale. On parle encore dans ce cas de "**container à JSP**" pour désigner un container (ou moteur) à servlets étendu qui prend en charge l'appel du compilateur de pages (si nécessaire, autrement dit si une nouvelle version du JSP est détectée). On s'en doute, si le client a entré "4" dans la page d'accueil, la page reçue aura l'aspect suivant (en utilisant Tomcat en local) :



Du point de vue du client, tout se passe comme s'il recevait une page HTML statique. On peut s'en convaincre en regardant le source dans le browser :

RepeteBonjour.jsp → page HTML

```
<HTML>
<HEAD>
<TITLE>Bonjour à répétition</TITLE>
</HEAD>
<BODY>
*** Accueil dans le monde des JSP ***
<p>
    Bonjour ! <BR>
    Bonjour ! <BR>
    Bonjour ! <BR>
    Bonjour ! <BR>
</P>
Cela nous fait 4 fois ...
</BODY>
</HTML>
```

Bien sûr, tout comme pour les servlets, il est plus alléchant d'appeler le JSP depuis une autre page HTML :

ACCUEIL.HTML

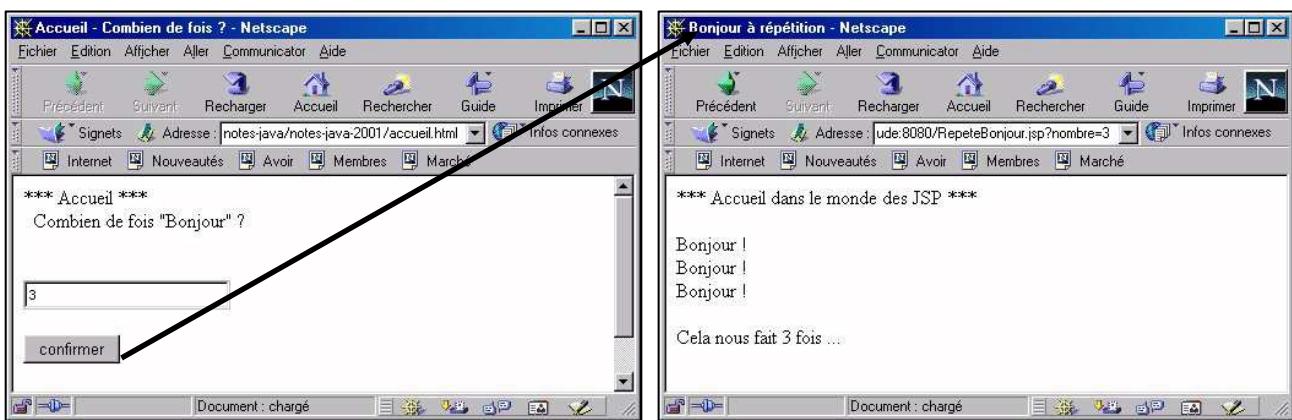
```
<HTML>
<HEAD>
    <META HTTP-EQUIV="Content-Type" CONTENT="text/html; charset=iso-8859-1">
    <META NAME="Author" CONTENT="Claude VILVENS">
    <META NAME="GENERATOR" CONTENT="Mozilla/4.04 [en] (Win95; I) [Netscape]">
    <TITLE>Accueil – Combien de fois ?</TITLE>
</HEAD>
```

```
<BODY>
*** Accueil ***
<BR>&nbsp;
Combien de fois "Bonjour" ?
<BR>&nbsp;

<form method="GET" action="http://claudie:8080/RepeteBonjour.jsp">
<P><input type="text" name="nombre" size=20></P>
<P><input type="submit" value="confirmer"></P>
</form>

<BR>&nbsp;
</BODY>
</HTML>
```

L'appui sur le bouton "confirmer" va envoyer la requête au serveur Web, lequel va se mettre à la recherche du JSP demandé :



4. Le cycle de vie d'un JSP

Que se passe-t-il derrière la scène, c'est-à-dire du côté du serveur ? On s'en doute, le moteur à servlets est toujours bien présent. Pour rappel, il consiste, pour l'essentiel, en une application Java tournant sur une machine virtuelle. Il devient un moteur à JSP à partir du moment où il a lancé une servlet particulière appelée le "**compilateur de pages**" à qui il sera fait appel à toute requête faisant intervenir un JSP.

Le compilateur de pages du moteur JSP va transformer la page JSP en une servlet Java

servlet analogue à celles dont nous avons abondamment parlé dans le chapitre XVII. Les éventuelles méthodes définies dans les scriptlets deviennent alors des méthodes de cette servlet. Ainsi, dans le cas de notre exemple, la servlet résultante est du genre (le détail ne nous intéresse guère ici) :

servlet correspondant à RepeteBonjour.jsp

```
package org.apache.jsp;

import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;

public final class RepeteBonjour_jsp extends org.apache.jasper.runtime.HttpJspBase
    implements org.apache.jasper.runtime.JspSourceDependent
{
    private static final JspFactory _jspxFactory = JspFactory.getDefaultFactory();
    private static java.util.List _jspx_dependants;
    private javax.el.ExpressionFactory _el_expressionfactory;
    private org.apache.AnnotationProcessor _jsp_annotationprocessor;
    public Object getDependants() { return _jspx_dependants; }
    public void _jspInit()
    {
        _el_expressionfactory = _jspxFactory.getJspApplicationContext(getServletConfig().
getServletContext()).getExpressionFactory();
        _jsp_annotationprocessor = (org.apache.AnnotationProcessor) getServletConfig().
getServletContext().getAttribute(org.apache.AnnotationProcessor.class.getName());
    }
    public void _jspDestroy() { }

    public void _jspService(HttpServletRequest request, HttpServletResponse response)
        throws java.io.IOException, ServletException
    {
        PageContext pageContext = null;
        HttpSession session = null;
        ServletContext application = null;
        ServletConfig config = null;
        JspWriter out = null;
        Object page = this;
        JspWriter _jspx_out = null;
        PageContext _jspx_page_context = null;

        try
        {
            response.setContentType("text/html;charset=UTF-8");
            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("\n"); out.write("\n"); out.write("\n");
out.write("<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
Transitional//EN\"\n");
out.write(" \\"http://www.w3.org/TR/html4/loose.dtd\">\n"); out.write("\n");
out.write("<html>\n"); out.write("   <head>\n");
out.write("<meta http-equiv=\"Content-Type\" content=\"text/html; charset=UTF-8\">\n");
out.write("      <title>Bonjour à répétition</title>\n");
out.write("   </head>\n"); out.write("   <body>\n");
int nbre=Integer.parseInt(request.getParameter("nombre"));
for (int i=0; i<nbre; i++)
{
    out.write("\n"); out.write("      Bonjour ! <br>\n"); out.write("      ");
}

out.write("\n"); out.write("   </body>\n"); out.write("</html>\n");
}
catch (Throwable t)
{
if (!(t instanceof SkipPageException)){
    out = _jspx_out;
    if (out != null && out.getBufferSize() != 0)
        try { out.clearBuffer(); } catch (java.io.IOException e) {}
    if (_jspx_page_context != null) _jspx_page_context.handlePageException(t);
}
} finally {
    _jspxFactory.releasePageContext(_jspx_page_context);
}
}
}
}

```

La lecture, aride, de ce code montre le code déjà présent dans le JSP et indique aussi, par exemple, qu'un objet session peut être géré. La classe **HttpJspBase**, dont seront dérivées toutes les servlets issues de la compilation d'un JSP, est une classe dérivée de la classique **HttpServlet** et se trouvant dans le package **org.apache.jasper.runtime**.

La servlet obtenue sera ensuite, sans surprise, compilée (du moins si il s'agit d'une nouvelle version) puis exécutée sur la machine virtuelle du serveur. A la première requête, la méthode **_jspInit()** sera exécutée. Ensuite, dans tous les cas, un thread sera alors lancé pour exécuter la méthode **_jspService()** de la servlet.

Remarque

L'attribut **workDir** de la balise **<Host>** du fichier de configuration de Tomcat **server.xml** spécifie le "répertoire de travail" pour les applications, c'est-à-dire notamment l'endroit où les servlets générées résultant des JSPs sont stockées. Il existe une valeur par défaut :

CATALINA_HOME/work/<nom_engine>/<nom_host>

- donc pour nous qui n'avons rien changé à la configuration originale :

C:\Program Files\Apache Software Foundation\Tomcat 6\work\Catalina\localhost

5. Servlets ou JSP ?

Autrement dit, par le travail du compilateur de pages, le fonctionnement d'un JSP se ramène au fonctionnement normal d'une servlet ! Et la question vient évidemment immédiatement à l'esprit : ***pourquoi ne pas se contenter des servlets ?***

Tout simplement parce que le développement de pages dynamiques est ressenti comme plus simple : la logique des applications, contenue dans les scriptlets, est bien distincte des tags HTML qui ne visent qu'à l'interface. Donc,

les JSP offrent une séparation bien nette entre la présentation et la logique de programmation

On peut d'ailleurs imaginer d'utiliser un ensemble de templates HTML dans lesquels il n'y plus qu'à placer le code "intelligent". Les modifications de présentation pourront donc être effectuées par des non-programmeurs (ce qui est tout le contraire des pages dynamiques fournies par des servlets), du moins si l'application visée n'est pas trop complexe – nous y reviendrons en fin de chapitre.

Avant d'illustrer la mise au point d'un JSP avec NetBeans et Tomcat, il nous faut encore cependant donner quelques précision sur les objets implicites de ces JSP ...

6. Les objets implicites

Afin de faciliter le travail des développeurs, les concepteurs des moteurs à JSP ont décidé que ce moteur fournirait un certain nombre d'objets prédefinis, dont il y a de grandes chances que ces développeurs aient besoin. Il s'agit, pour ce qui nous intéresse directement, de :

objet implicite	classe instanciée ou interface implémenté	package correspondant
request	HttpServletRequest	javax.servlet.http
response	HttpServletResponse	javax.servlet.http
session	HttpSession	javax.servlet.http
config	ServletConfig	javax.servlet
out	JspWriter	javax.servlet.jsp
application	ServletContext	javax.servlet
pageContext	PageContext	javax.servlet.jsp
exception	Throwable	java.lang

D'emblée, on aura reconnu les objets classiques de la programmation des servlets : request, response et, éventuellement, session et config. Ils jouent le même rôle que pour les servlets classiques.

L'objet ***application*** prend son sens si l'on sait que les moteurs à JSP groupent celles-ci en fonction de leur URL : le premier nom de répertoire qui y apparaît désigne implicitement l'application Web. Autrement dit, **tous les JSP se trouvant dans le même répertoire appartiennent à la même application Web**. L'objet implicite application ne sert qu'à matérialiser cette notion, en apportant notamment la possibilité d'associer des objets à l'application, objets partageables par les différents JSP.

L'objet ***pageContext*** fournit un accès à tous les objets implicites ainsi qu'à leurs attributs. Il permet surtout de transférer le contrôle de la page courante à une autre page.

On peut ainsi constater l'existence d'un sous-package javax.servlet.jsp fournissant en même temps une classe JspWriter dont l'objet implicite **out** est une instance. La classe JspWriter est définie dans ce package et dérive de la classe Writer de java.io; c'est un flux de sortie connecté sur la page HTML qui va être renvoyée comme réponse par le JSP. Il s'agit en fait d'une espèce de PrintWriter (comme System.out), mais orienté JSP et utilisant le mécanisme des exceptions. Il comporte essentiellement :

- ◆ des méthodes d'affichage, donc plus exactement de sortie vers la future page HTML, comme par exemple :

```
public abstract void print(java.lang.String s) throws java.io.IOException  
public abstract void println(java.lang.String x) throws java.io.IOException
```

- ◆ des méthodes de gestion des buffers , comme

```
public abstract void flush() throws java.io.IOException  
public abstract void newLine() throws java.io.IOException
```

(ce que fait exactement cette méthode dépend de la définition de la propriété line.separator de la classe System)

Donc, on peut remplacer dans un JSP :

<p>Vous êtes donc :

par :

```
<% out.println("<p>Vous êtes donc :"); %>
```

Voyons à présent comment développer un JSP en pratique.

7. Un classique : le traitement d'un formulaire

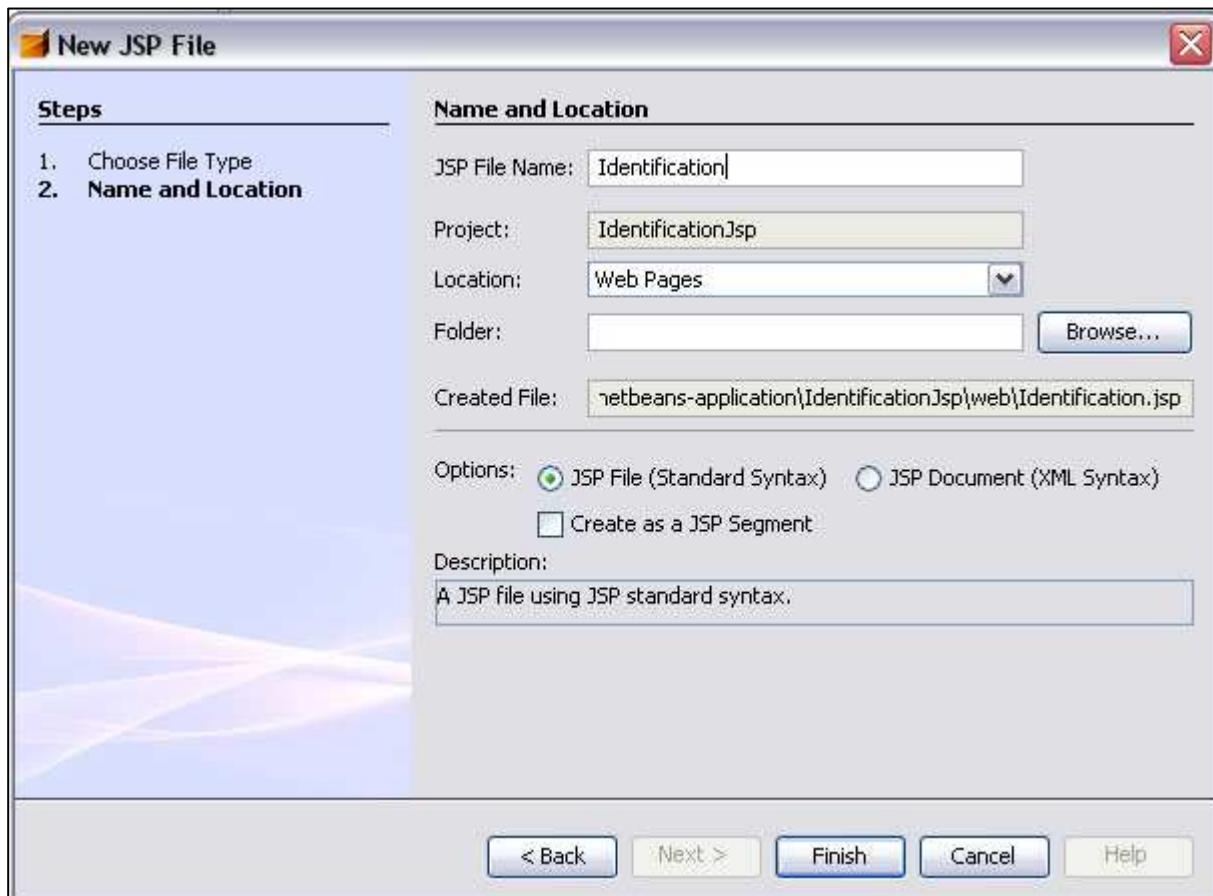
De manière assez peu originale, considérons, comme pour les servlets, un JSP dont le rôle est de traiter une requête cliente introduite au moyen d'un formulaire. Pour fixer les idées, disons, provisoirement, que le formulaire possèdera trois zone d'entrée : nom, prenom et pwd.

7.1 La création du JSP avec NetBeans

Nous créerons la page HTML contenant le formulaire plus tard. Nous allons d'abord créer le JSP sous NetBeans au sein d'une application Web préalablement créée. Un simple clic droit sur le nœud Web Pages (car un JSP est une page Web un peu particulière) nous permet de choisir :

New → JSP ...

qui conduit à un wizzard classique :



Le résultat obtenu, fort encombré de concepts évolués, ressemble à ceci :

Identification.jsp (code généré)

```
<%@page contentType="text/html"%>
<%@page pageEncoding=UTF-8%>
<%-- The taglib directive below imports the JSTL library. If you uncomment it,
you must also add the JSTL library to the project. The Add Library... action
on Libraries node in Projects view can be used to add the JSTL 1.1 library.
--%>
<%-->
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix=c%>
<%-->

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

<html>
  <head>
    <meta http-equiv=Content-Type content="text/html; charset=UTF-8">
    <title>JSP Page</title>
  </head>
  <body>
```

```
<h1>JSP Page</h1>

<%--  
This example uses JSTL, uncomment the taglib directive above.  
To test, display the page like this: index.jsp?sayHello=true&name=Murphy  
--%>  
<%--  
<c:if test="${param.sayHello}">  
    <!-- Let's welcome the user ${param.name} -->  
    Hello ${param.name}!  
</c:if>  
--%>  
  
</body>  
</html>
```

On peut à présent modifier le texte du JSP à sa convenance. Nous allons donc

1) récupérer les données entrées dans le formulaire comme dans une servlet, soit au moyen de la méthode

public abstract String **getParameter**(String name)

appelée par l'objet implicite request.

2) insérer la date courante, tant dans la page HTML que dans son texte. Il faut pour cela

♦ **importer les packages** nécessaires par une directive page comportant l'attribut import :

```
<%@page import="java.util.*" %>  
<%@page import="java.text.*" %>
```

♦ **déclarer les variables** habituelles pour obtenir une date au format local, donc ici en français:

```
<%! Date maintenant = new Date(); %>  
<%! String laDate = DateFormat.getDateInstance(DateFormat.FULL,  
DateFormat.FULL,Locale.FRANCE).format(maintenant); %>
```

Ici, pour rappel, il ne s'agit pas d'une scriptlet mais d'une déclaration : les variables maintenant et laDate ne sont pas des variables locales, mais bien des variables d'instance de la servlet résultant de la compilation de la page JSP.

♦ **insérer la date**, à la fois dans le page et comme commentaire (au sens HTML) dans celle-ci :

```
<!-- Page demandée le <%=laDate %> -->  
Nous sommes le <%=laDate %> !!!
```

3) **ajouter un texte explicatif** accessible au moteur, par exemple dans un but de copyright :

```
<%@page info="(c) Claude Vilvens - 9/2009" %>
```

Cette information peut être récupérée au moyen de la méthode :

```
public abstract String getServletInfo()
```

Cela peut donc finalement donner :

Identification.jsp (code modifié)

```
<%@page language="java" %>
<%@page contentType="text/html; charset=ISO-8859-1"%>
<%@page pageEncoding="UTF-8"%>
<%@page info="(c) Claude Vilvens - 8/2007" %>
<%@page import="java.util.*" %>
<%@page import="java.text.*" %>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
 "http://www.w3.org/TR/html4/loose.dtd">
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>Réponse à l'identification</title>
  </head>
  <body>

    <%! Date maintenant = new Date(); %>
    <%! String laDate = DateFormat.getDateInstance(
      DateFormat.FULL,DateFormat.FULL,Locale.FRANCE).
      format(maintenant); %>

    <!-- Page demandée le <%=laDate %> -->

    <h3>Bienvenue sur notre site coquin !</h3>

    Nous sommes le <%=laDate %> !!!
    <p>Vous êtes donc : <p>
    <% String nom = request.getParameter("nom");
      String prenom = request.getParameter("prenom");
      String motDePasse = request.getParameter("pwd");
      %>

    <%= nom %><%=prenom%> (**<%= motDePasse%> ***)>

    <p>(généré par <%=getServletInfo() %>)

  </body>
</html>
```

Remarque

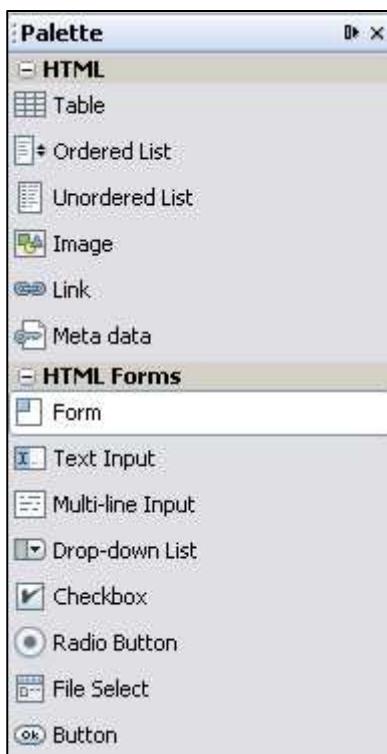
On peut visualiser le contenu de la servlet générée par le compilateur au moyen du choix View Servlet dans le menu obtenu d'un clic droit sur l'objet Identification.

7.2 La création de la page formulaire

Nous pourrions évidemment créer à la main la page HTML permettant d'appeler ce JSP. Mais NetBeans facilite grandement le travail. Avec

New → HTML File

nous obtenons une page Web de base (nous l'avons appelée AccueilIdentification.html) accompagnée d'une palette de composants HTML qui ne demandent qu'à être "dropés" :



D'emblée, c'est le tag Form qui va se montrer fort pratique car un drop dans la page fait apparaître la boîte de dialogue :



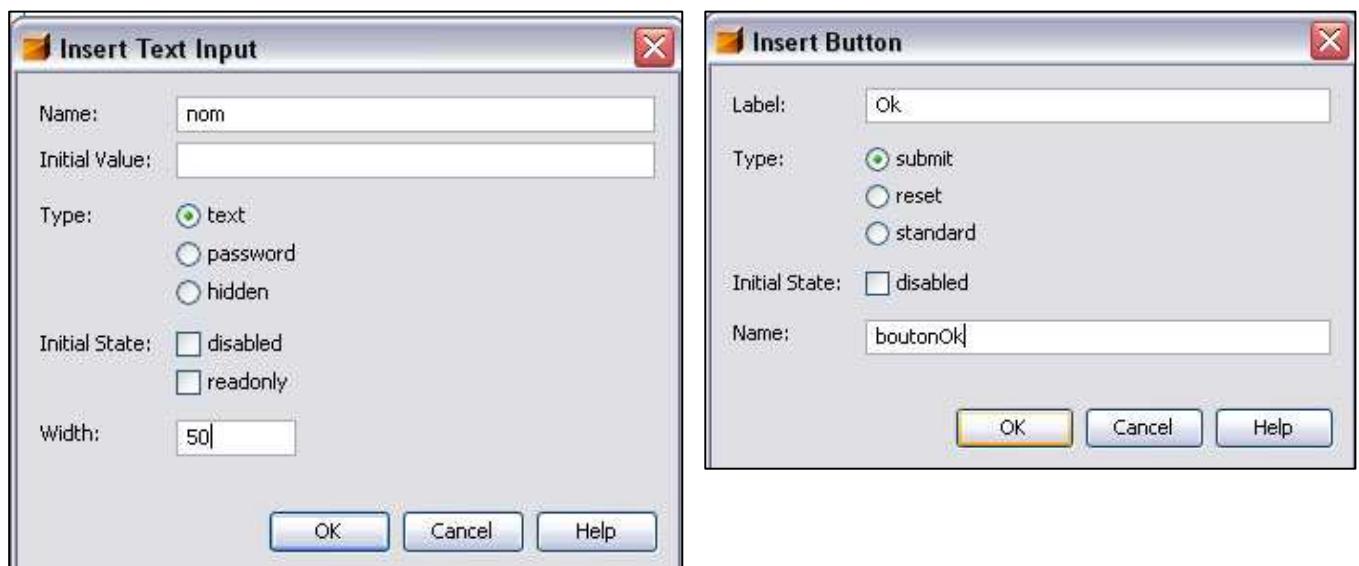
L'action doit désigner notre JSP, que nous pouvons choisir dans la liste des JSPs de l'application Web, liste obtenue par appui sur le bouton Browse :



A ce stade, nous disposons alors du tag :

```
<form name="FormulaireIdentification" action="Identification.jsp"></form>
```

Il ne nous reste plus qu'à placer les différentes zones du formulaire :



On parvient ainsi à :

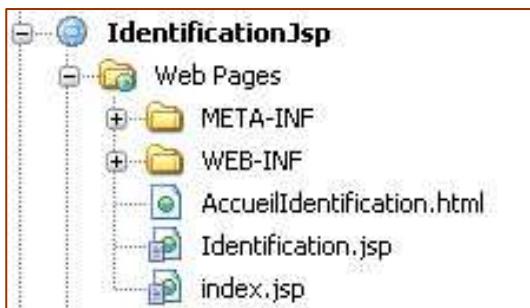
AccueilIdentification.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">

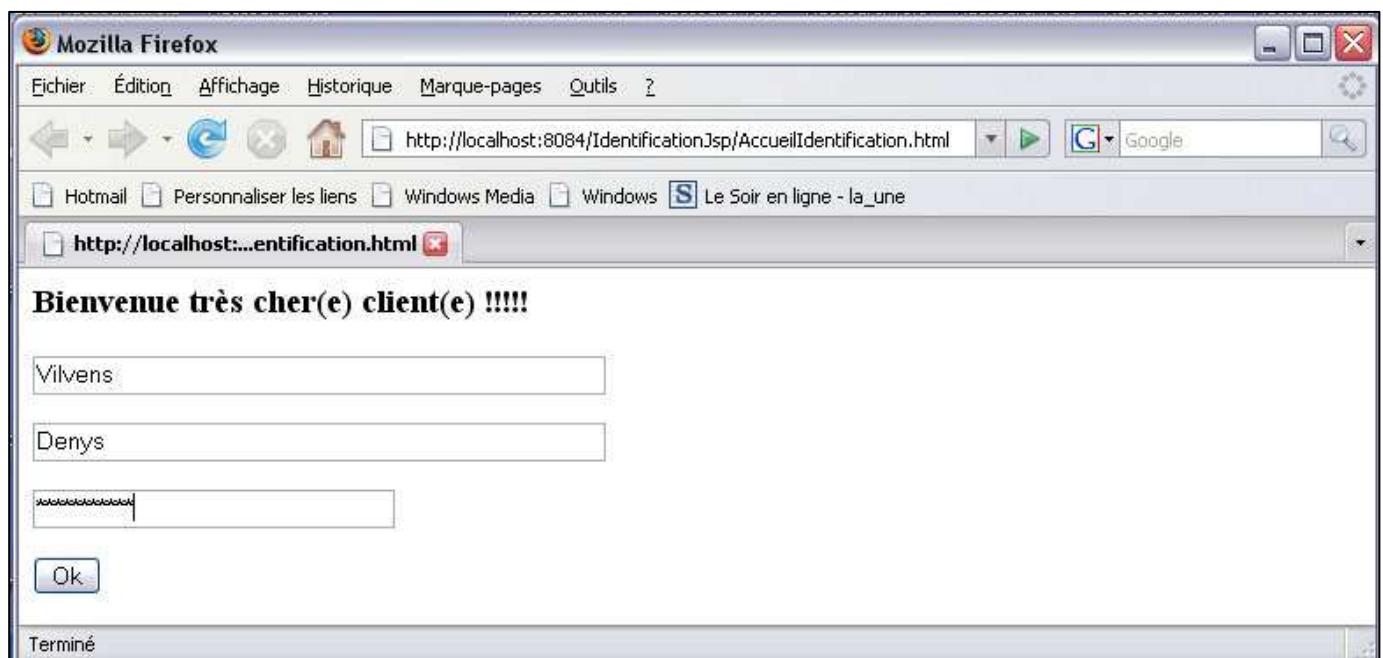
<html>
<head>
<title></title>
</head>
<body>
<H3>Bienvenue très cher(e) client(e) !!!!! </H3><p>
<form name="FormulaireIdentification" action="Identification.jsp">
<input type="text" name="nom" value="" size="50" /><p>
<input type="text" name="prenom" value="" size="50" /><p>
<input type="password" name="pwd" value="" size="30" /><p>
<input type="submit" value="Ok" name="boutonOk" /><p>
</form>
</body>
</html>
```

7.3 L'exécution de l'application en local avec Tomcat intégré à NetBeans

Notre application Web comporte donc à présent la page de démarrage et le JSP.



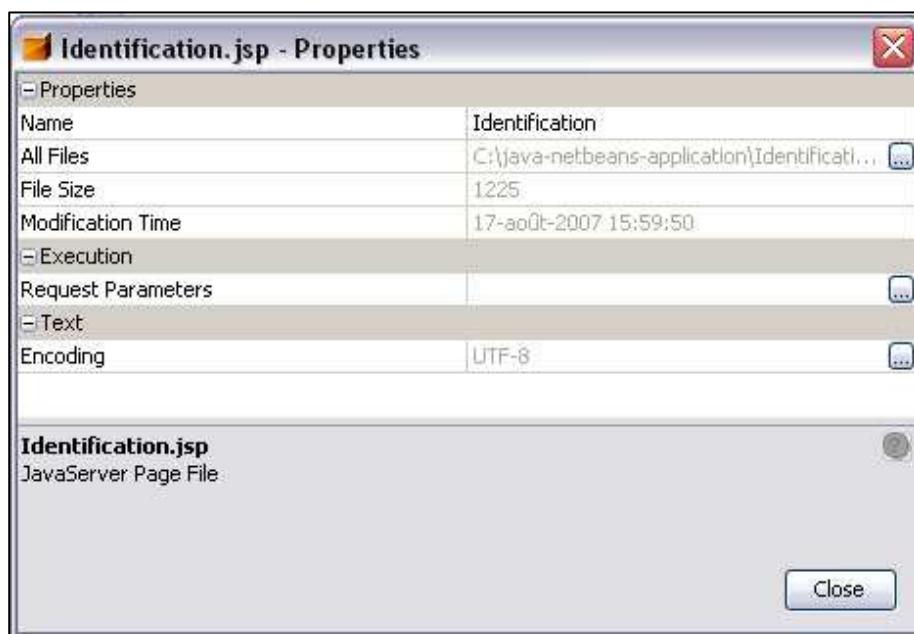
Il suffit de lancer l'exécution de la page HTML :



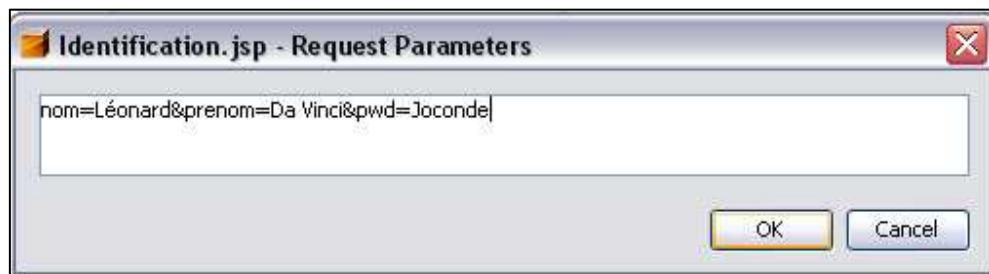


7.4 Le test direct d'un JSP

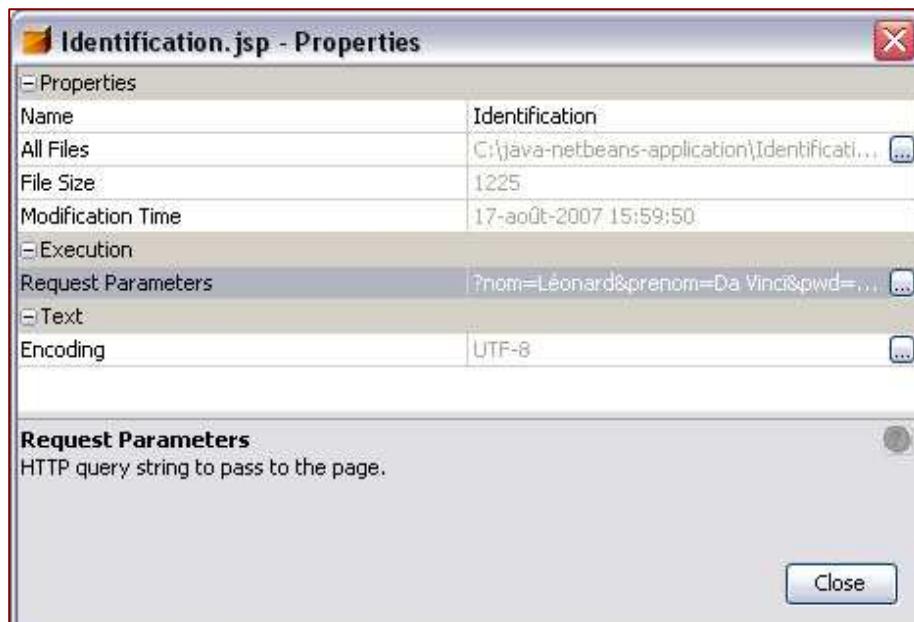
Pour tester notre JSP au sein de NetBeans sans même devoir créer une page HTML, on peut d'abord spécifier la valeur des paramètres du formulaire que l'on prévoit pour l'appel en éditant les Properties de l'objet Identification.jsp :



C'est le champ Request Parameters qui nous intéresse :



ce qui donne :



L'exécution directe du JSP va provoquer la compilation de page en une servlet (si ce n'est pas déjà fait), la mise en place de l'application Web, le lancement du Tomcat intégré et finalement l'exécution de l'application :

Bienvenue sur notre site coquin !

Nous sommes le samedi 18 août 2007 07 h 44 CEST !!!

Vous êtes donc :

Léonard Da Vinci (*** Joconde ***)

(généré par (c) Claude Vilvens - 8/2007)

Terminé

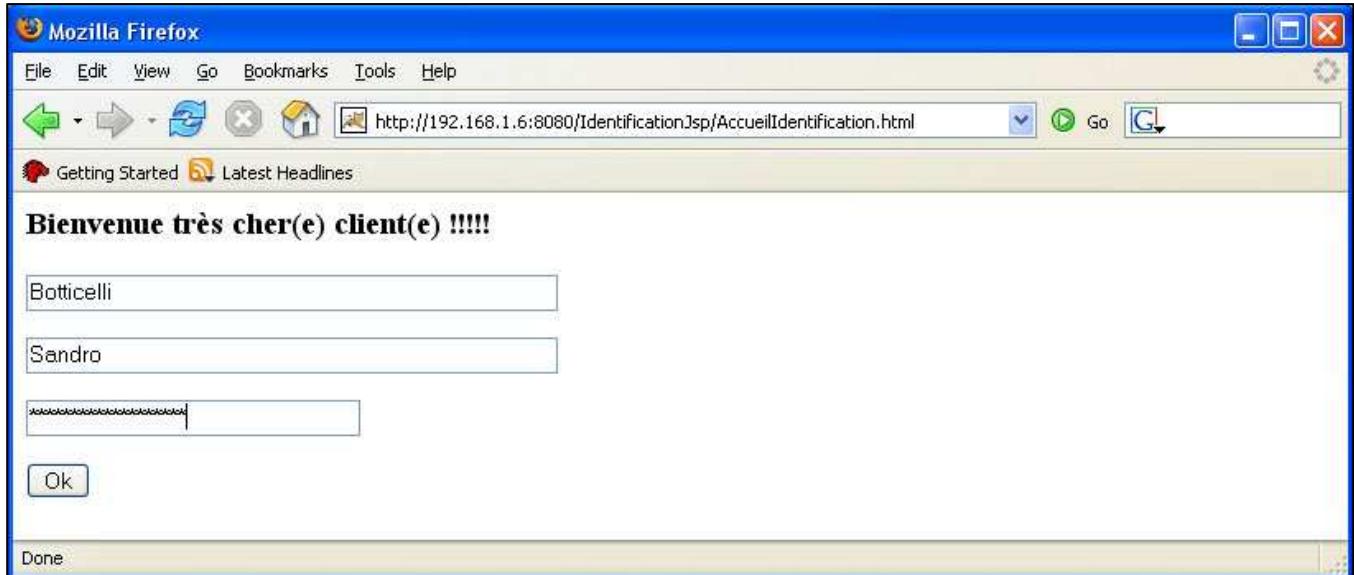
Il faut aussi remarquer que la servlet résultante gère une session HTTP créée par défaut : le HTTP Monitor nous le montre :

Session properties	Value
Session ID	49122D1AB75B8B11F95FA8AB4610A4FC
Created	samedi 18 août 2007 7:49
Last accessed	samedi 18 août 2007 7:49
Max inactive interval	1800

Session attributes after the request: none

7.5 L'exécution de l'application avec Tomcat standalone

1) Le plus simple est évidemment de procéder comme pour les applications Web à base de servlets : le war trouvé dans le répertoire dist est placé dans le répertoire webapps de Tomcat et il n'y a plus qu'à agir. Ainsi, à partir d'une machine cliente distante :



2) Si (pour une raison obscure – mais tous les goûts sont dans la nature ...) on ne souhaite pas travailler avec un war, il suffit de placer les fichiers Identification.jsp et AccueilIdentification.html, trouvés dans le répertoire **web** de notre projet NetBeans, dans le **C:\Program Files\Apache Software Foundation\Tomcat 5.5\webapps\ROOT** de la machine serveur. Le résultat est bien sûr parfaitement le même :

Mozilla Firefox

File Edit View Go Bookmarks Tools Help

Getting Started Latest Headlines

Bienvenue très cher(e) client(e) !!!!

Done



Réponse à l'identification - Mozilla Firefox

File Edit View Go Bookmarks Tools Help

Getting Started Latest Headlines

Bienvenue sur notre site coquin !

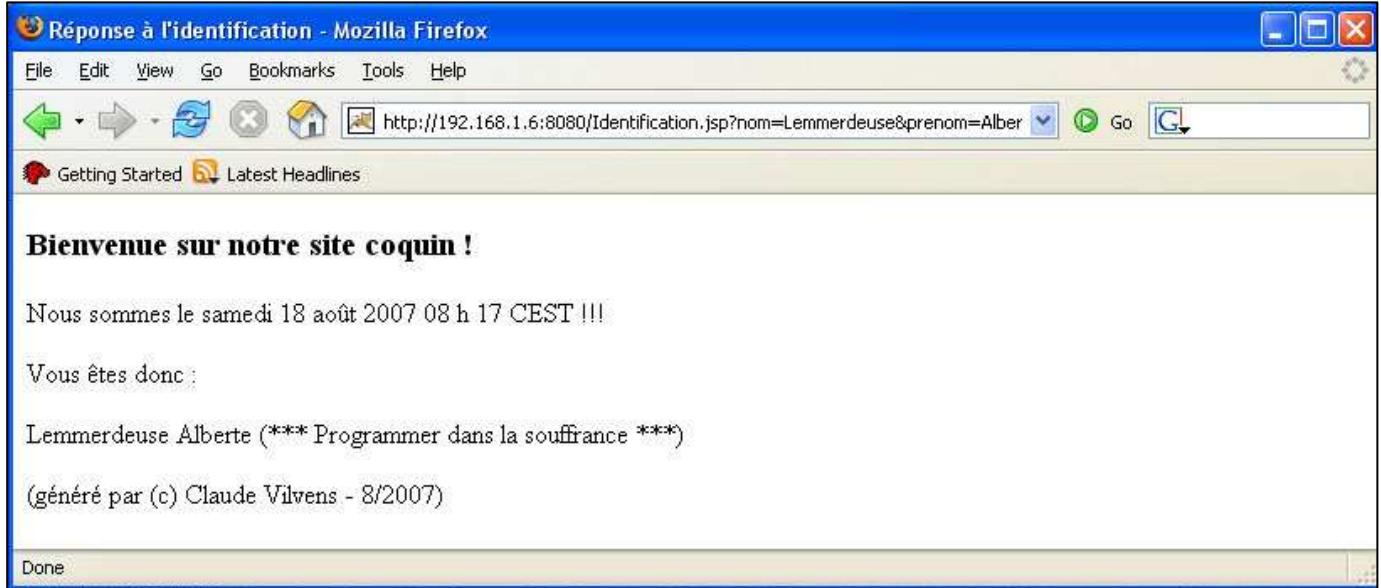
Nous sommes le samedi 18 août 2007 08 h 17 CEST !!!

Vous êtes donc :

Lemmerdeuse Alberte (** Programmer dans la souffrance ***)

(généré par (c) Claude Vilvens - 8/2007)

Done



Remarque

On peut trouver plus propre de créer des sous-répertoires dédiacés :

- ◆ un répertoire **JSP** (par exemple) pour contenir les Java Server Pages;
- ◆ un répertoire **public** (par exemple) pour les pages HTML.

Ceci apportera comme seules différences que :

- l'appel du JSP comportera en plus le répertoire JSP :

```
<form method="get" action="http://192.168.1.6:8080/JSP/Identification.jsp">
```

- la page du formulaire est invoquée avec la spécification du répertoire public :

```
http://192.168.1.6:8080/public/AccueilIdentification.html
```

7.6 L'utilisation avec Tomcat sur un serveur Linux

Dans le contexte Linux de la machine U2 de l'InPrES, il nous suffit soit de placer

- ◆ notre fichier Identification.jsp dans le répertoire **jsp**;
- ◆ la page HTML éventuelle dans le répertoire **public_html**.

Si le path défini dans la balise context de server.xml est, par exemple, webdev2 :

```
<Context    path="/webdev2"  
          docBase="/home/vilvens/webdev"  
          debug="0" reloadable="true">  
    <Logger className="org.apache.catalina.logger.FileLogger"  
           prefix="localhost_pw_log." suffix=".txt"  
           timestamp="true"/>  
</Context>
```

il suffit alors d'utiliser, depuis le browser ou depuis une page HTML, l'URL :

<http://u2:8180/webdev2/jsp/Identification.jsp?nom=Vil&prenom=Cloclo&pwd=beau>

pour appeler le JSP directement et obtenir les mêmes résultats qu'en local. Pour rappel, le path doit être unique au niveau de server.xml.

Dans le contexte de la machine Indochine de l'InPrES, il suffit d'utiliser le war comme sous Windows.

8. Les déclarations de variables et de méthodes

Il est possible, comme on l'a déjà dit, de déclarer des variables et des méthodes dans un JSP en utilisant les tags `<%! ... %>`. *Ces variables et méthodes sont alors des membres de la classe servlet* résultant de la compilation du JSP en servlet. A titre d'exemple, le JSP suivant va réaliser un accès à la table produits d'une base de données ACCESS "marie-inpres", dont le nom DSN est "marie". La table en question ressemble à ceci :

table produits

codePr	descriptif	taille1	taille2	tol_t1	tol_t2	proprietes
APHRO700	céleri	22	19	2	2	effet garanti
ASP69	asperges longues	25	2	1	1	gare aux dents
CH2000	champignons	15	5	1	0,8	ça vaut l'herbe
PP1000	petits pois billes	4	-1	0,5	-1	instestinal

tandis que le JSP s'écrit :

ProduitsMarie.jsp

```
<%@page contentType="text/html"%>  
  
<%@page import="java.sql.*"%>  
<%@page import="java.util.*"%>
```

```
<%@page import="java.text.*" %>

<%! private String maDate= chaineDateDuJour(); %>

<html>
<head><title>JSP et JDBC</title></head>

<body>
<H3>MARIE INPRES !!!</H3> (<%=maDate %>)
<BR><B>... La qualité de vos légumes ...</B>
<HR>

<% Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
  Connection con = DriverManager.getConnection("jdbc:odbc:marie","","");
  Statement instruc = con.createStatement();
  ResultSet rs = instruc.executeQuery("select * from produits");
  int cpt = 0;

  while (rs.next())
  {
<%>
    <%cpt++;%>
    Produits : <%= rs.getString("codePr") %><BR>
    Descriptif : <%= rs.getString("descriptif") %><BR>
    Propriétés : <%= rs.getString("proprietes") %><BR>
    <%= chaineTaille (rs.getFloat("taille1")) %><BR>
    <HR>
<%}>
  %>

<%! public String chaineTaille(float taille)
  {
    return "Taille = " + new Float(taille).toString() + "cm";
  }
%>

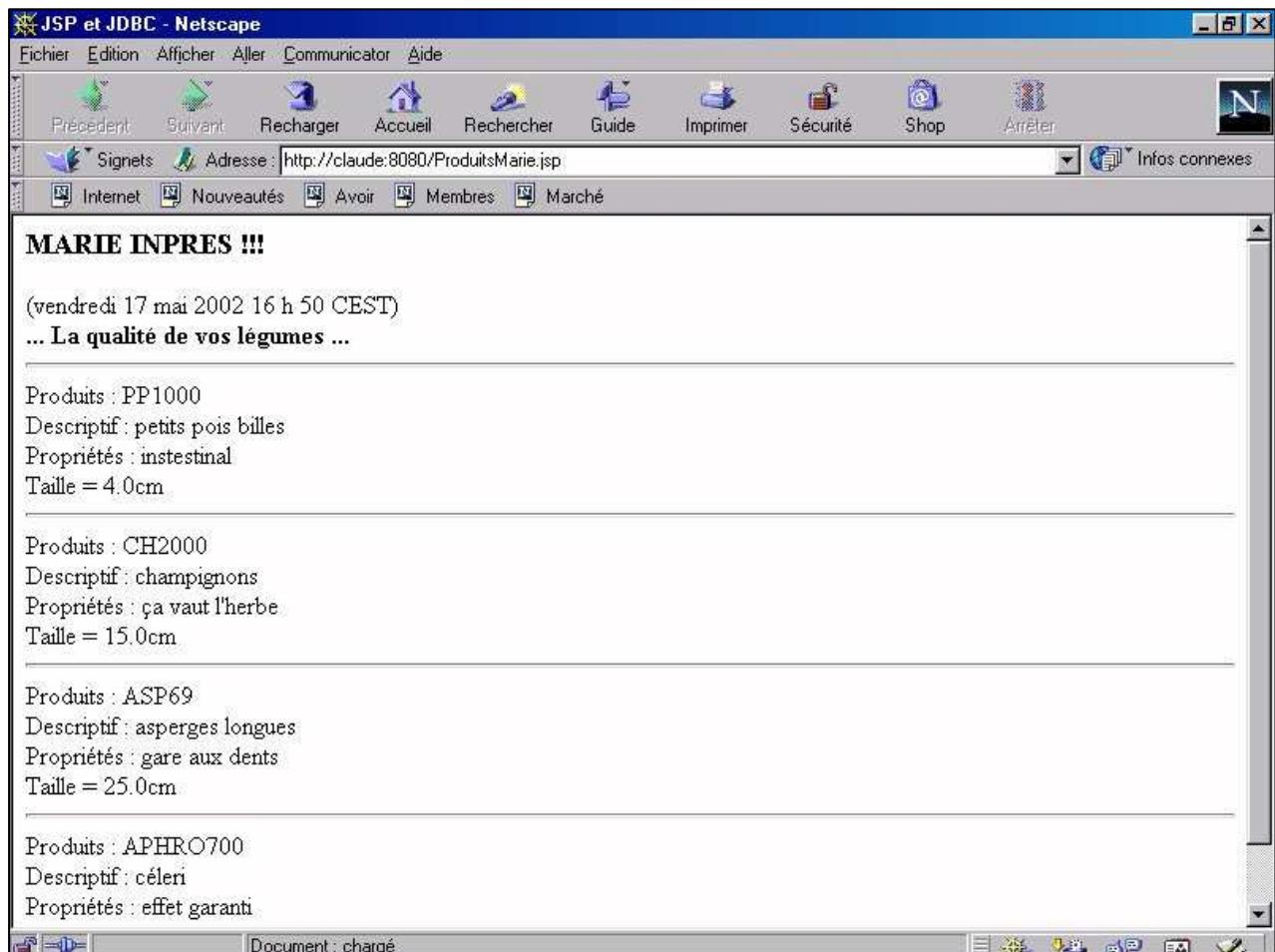
<%! private String chaineDateDuJour()
  {
    java.util.Date maintenant = new java.util.Date();
    return DateFormat.getDateInstance(DateFormat.FULL,DateFormat.FULL,
      Locale.FRANCE).format(maintenant);
  }
%>

</body>
</html>
```

Plusieurs petites choses sont à remarquer :

- ◆ l'accès JDBC, tout à fait classique, si ce n'est que tout se fait au sein d'une scriptlet;
- ◆ la définition de deux méthodes, chaineTaille() et chaineDateDuJour(), ainsi que la définition de la variable membre maDate, toutes ces définitions débutant par "<%!";
- ◆ l'appel des méthodes, l'une pour initialiser une variable, l'autre dans une expression.

Un exemple d'exécution donne :



9. L'utilisation des Java Beans

9.1 Trois actions fondamentales

On l'a dit, le grand intérêt des JSP est de séparer la présentation de la logique fonctionnelle. Dans cet esprit, et dans celui de Java, il est très naturel de penser à confier les traitements à des composants comme les Java Beans. Notamment pour utiliser ces derniers, les JSP fournissent des balises particulières qui sont des "**actions**" : il s'agit de balises permettant de spécifier l'utilisation d'un élément extérieur (un bean, un plugin, un autre JSP) et/ou d'effectuer une tâche selon des informations obtenues au moment où le JSP est accédé par le client. De manière générale, une action (encore appelée couramment "balise") suit la syntaxe :

```
<nom_action attribut=valeur [attribut=valeur]>  
corps_action  
</nom_action>
```

où

nom_action := nom_bibliothèque:nom_action_dans_bibliothèque

Une bibliothèque de base est celle dont le préfixe est "jsp" : c'est la bibliothèque standard des JSPs. Mais, en fait, on peut créer ses propres librairies de balises (voir plus loin), ce que les grands constructeurs logiciels n'ont pas manqué de faire (<sun: ...>, <ora: ...>, <struts: ...>).

Il convient de bien distinguer ces actions des directives :

- ◆ une directive est simplement utilisée lors de la "compilation de page", c'est-à-dire lors de la génération de la servlet correspondant au JSP;
- ◆ une action représente une *action dynamique* qui a lieu lors de l'exécution de cette servlet.

Dans le cas qui nous occupe, c'est-à-dire l'utilisation standard, notre objectif est d'utiliser un bean en usant de telles balises plutôt que de faire intervenir du code dans un scriptlet. Trois actions essentielles sont disponibles, avec un certain nombre d'attributs dont nous ne citerons que les principaux :

1) <jsp:useBean id="<nom du bean>" class="<nom de la classe>" scope="<portée>" />

ou

```
<jsp:useBean id="<nom du bean>" class="<nom de la classe>" scope="<portée>" >
    <code d'initialisation>
</jsp:usebean>
```

Rend le composant précisé disponible dans la page avec :

- ◆ nom du bean : identificateur de l'objet instanciant le bean;
- ◆ nom de la classe : le nom de la classe du bean est court ou complet selon qu'une directive <@ page import ...> existe ou pas;
- ◆ portée : la durée de vie du composant; valeurs possibles = page, request, session ou application – par défaut, la durée de vie est limitée à la page;
- ◆ code d'initialisation : on peut ainsi initialiser les propriétés du bean au moyen de la balise <jsp:setProperty>.

Plus précisément, un bean du nom donné est recherché dans la portée spécifiée (par exemple, la session); si il n'est pas trouvé, une nouvelle instance de la classe indiquée est créée et placée dans l'objet matérialisant la portée (donc, par exemple, l'objet session).

2) <jsp:getProperty name="<nom du bean> property=<nom de propriété>/>

Permet d'obtenir la valeur de la propriété spécifiée du bean et de l'ajouter à la page réponse – cette balise produit donc un contenu visible dans la page HTML résultante du JSP. Mieux encore, il est possible d'insérer cette balise dans une balise HTML, ce qui permet de réaliser des pages à l'aspect particulièrement dynamique. Il convient encore de remarquer que les valeurs accessibles à travers cette balise sont toujours converties en objets String par le moteur à JSP.

3) <jsp:setProperty name="<nom du bean> property=<nom de propriété> value="<valeur>"/>

Permet évidemment de modifier la valeur de la propriété spécifiée du bean, à condition bien sûr que la propriété du bean ne soit pas read-only (existence d'une méthode setXXX). Lorsque les noms de propriétés sont les mêmes que ceux des paramètres d'une requête,

l'attribut property peut prendre la valeur "*" pour que les propriétés du bean soient initialisées en une seule fois avec ces paramètres.

A nouveau, les valeurs utilisées à travers cette balise sont toujours converties en objets String par le moteur à JSP (celui-ci invoque donc la méthode `toString()` pour tous les objets).

Remarques

- 1) Il est aussi possible
 - ◆ de créer un bean dans une scriptlet de manière tout à fait conventionnelle, c'est-à-dire en utilisant un constructeur; dans ce cas, cependant, le bean ne peut pas être utilisé dans une balise;
 - ◆ d'utiliser la valeur d'une propriété d'un bean dans une scriptlet d'une manière tout à fait normale, au moyen de la méthode associée `getXXX()`.
- 2) Il n'est pas possible de gérer au sein des balises les événements propres aux beans.
- 3) Il existe d'autres actions à usage moins courant. Citons cependant deux actions relatives à l'exécution d'une applet dans la page dynamique résultant de l'appel du JSP :

```
<jsp:plugin type="applet" code=<nom de la classe>" codebase=<URL du répertoire du fichier class, relativement au répertoire contenant la page>" jreversion="1.5" />
```

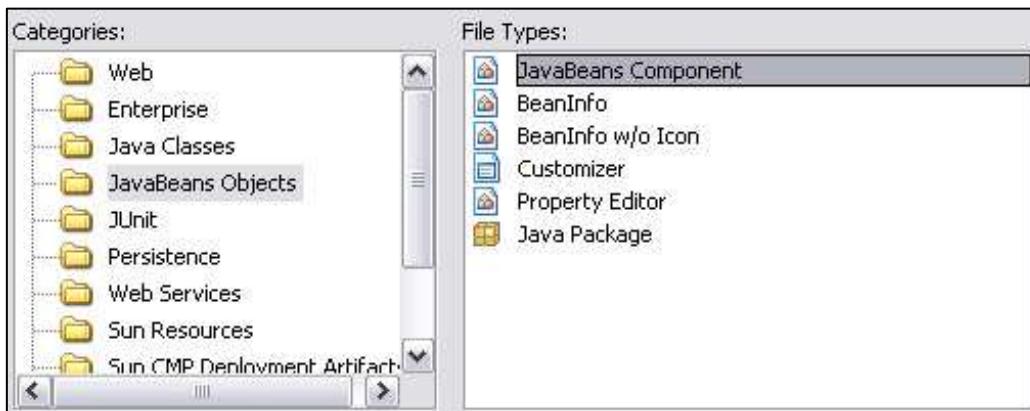
ou (si l'applet a des paramètres)

```
<jsp:plugin type="applet" code=<nom de la classe>" codebase=<URL du répertoire du fichier class, relativement au répertoire contenant la page>" jreversion="1.5"/>
<jsp:params>
    <jsp:param name=<nom d'un paramètre d'applet> value=<valeur du paramètre> >
</jsp:params>
</jsp:plugin>
```

Un attribut facultatif "archive" permet de charger le jar contenant les différents fichiers class associés à une applet.

9.2 Un bean pour un JSP

Nous allons utiliser dans un JSP, appelé `WelcomeUser`, un bean `PreferencesWeb` qui a pour rôle de contenir les préférences d'un utilisateur du Web en termes de couleurs de fond et de tracé, de langue et d'unité monétaire : ce sont ses propriétés. Pour cela, dans le noeud Source Packages de notre projet NetBeans, nous créons un package `Preferences`, puis à l'intérieur de celui-ci le Java Bean en question :



Ceci va nous donner un "template" de Java Bean :

PreferencesWeb.java (code généré)

```
/*
 * PreferencesWeb.java
 */
package Preferences;

import java.beans.*;
import java.io.Serializable;

/**
 * @author Vilvens
 */

public class PreferencesWeb extends Object implements Serializable
{
    public static final String PROP_SAMPLE_PROPERTY = "sampleProperty";
    private String sampleProperty;

    private PropertyChangeSupport propertySupport;

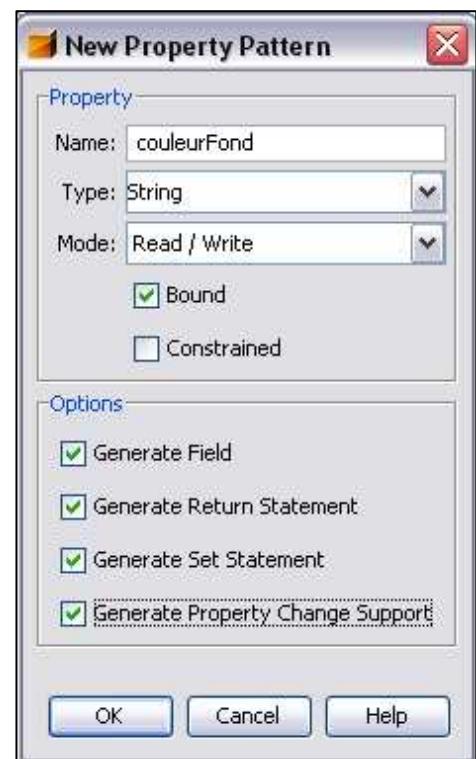
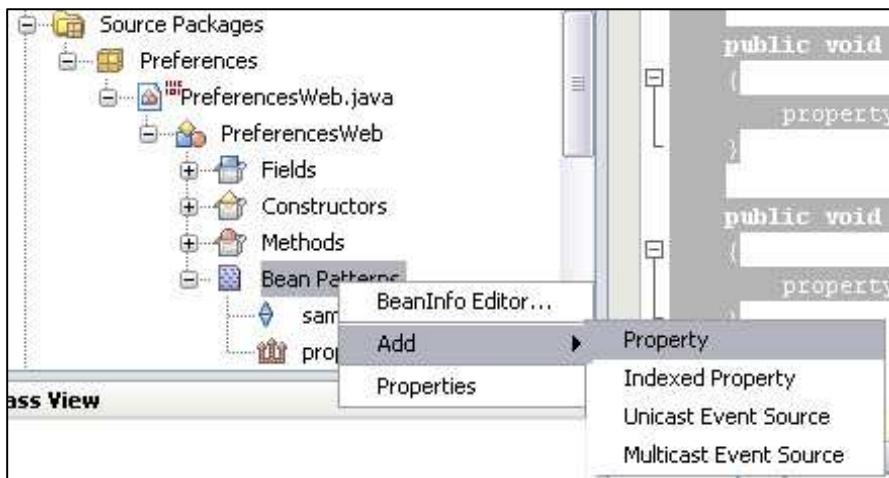
    public PreferencesWeb()
    {
        propertySupport = new PropertyChangeSupport(this);
    }

    public String getSampleProperty()
    {
        return sampleProperty;
    }
    public void setSampleProperty(String value)
    {
        String oldValue = sampleProperty;
        sampleProperty = value;
        propertySupport.firePropertyChange(PROP_SAMPLE_PROPERTY, oldValue,
            sampleProperty);
    }

    public void addPropertyChangeListener(PropertyChangeListener listener)
    {
        propertySupport.addPropertyChangeListener(listener);
    }

    public void removePropertyChangeListener(PropertyChangeListener listener)
    {
        propertySupport.removePropertyChangeListener(listener);
    }
}
```

Il nous reste à tailler ce bean selon nos désirs :



pour obtenir finalement (seules deux propriétés, couleurFond et uniteMonetaire ont été liées) :

PreferencesWeb.java (code modifié)

```
/*
 * PreferencesWeb.java
 */

package Preferences;

import java.beans.*;
import java.io.Serializable;
import java.awt.*;

/**
 * @author Vilvens
 */

public class PreferencesWeb extends Object implements Serializable
{
    private PropertyChangeSupport propertySupport;

    public PreferencesWeb()
    {
        couleurFond = "blue"; couleurTrait = "black";
        langue = "FR"; uniteMonetaire = "EUR";
        propertySupport = new PropertyChangeSupport(this);
    }
}
```

```

public void addPropertyChangeListener(PropertyChangeListener listener)
{ propertySupport.addPropertyChangeListener(listener); }
public void removePropertyChangeListener(PropertyChangeListener listener)
{ propertySupport.removePropertyChangeListener(listener); }

private String couleurFond;
public String getCouleurFond()
{
    return this.couleurFond;
}
public void setCouleurFond(String couleurFond)
{
    String oldCouleurFond = this.couleurFond;
    this.couleurFond = couleurFond;
    propertySupport.firePropertyChange ("couleurFond", oldCouleurFond, couleurFond);
}

private String couleurTrait;
public String getCouleurTrait()
{ return this.couleurTrait; }
public void setCouleurTrait(String couleurTrait)
{ this.couleurTrait = couleurTrait; }

private String langue;
public String getLangue()
{ return this.langue; }
public void setLangue(String langue)
{ this.langue = langue; }

private String uniteMonetaire;
public String getUniteMonetaire()
{ return this.uniteMonetaire; }
public void setUniteMonetaire(String uniteMonetaire)
{
    String oldUniteMonetaire = this.uniteMonetaire;
    this.uniteMonetaire = uniteMonetaire;
    propertySupport.firePropertyChange ("uniteMonetaire", oldUniteMonetaire,
        uniteMonetaire);
}
}

```

9.3 Un JSP qui utilise un bean

En tête de document JSP, nous allons préciser que nous utilisons une instance thePreferences de la classe Preferences.PreferencesWeb (tag jsp:usebean). Dans la balise body, nous allons récupérer la couleur de fond, propriété du bean en question (tag jsp:getProperty). De même, nous allons récupérer l'unité monétaire (toujours avec un tag jsp:getProperty). Cela donne :

WelcomeUser.jsp

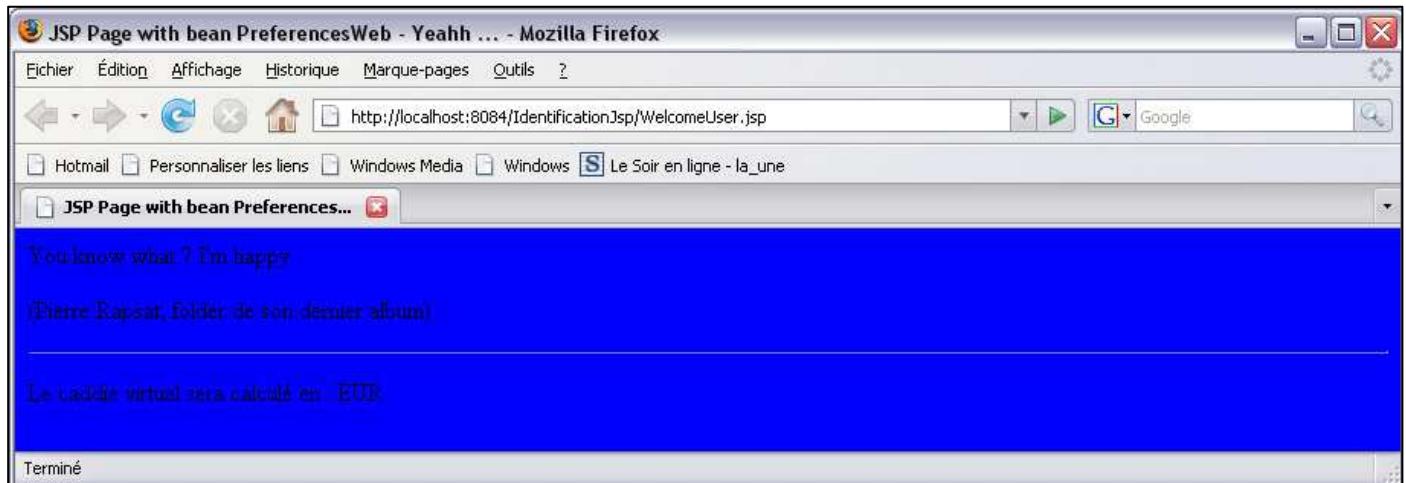
```
<%@page language="java" %>
<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>

<jsp:useBean id="thePreferences" scope="page" class="Preferences.PreferencesWeb" />

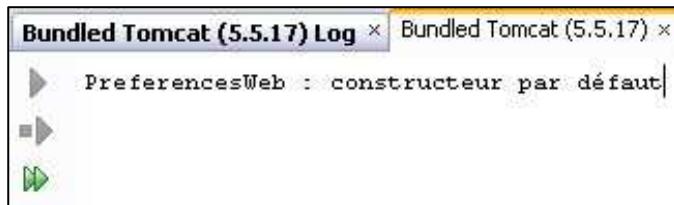
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
 "http://www.w3.org/TR/html4/loose.dtd">

<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>JSP Page with bean PreferencesWeb - Yeahh ... </title>
  </head>
  <body bgcolor=<jsp:getProperty name="thePreferences" property="couleurFond" />>
  You know what ? I'm happy
  <p>(Pierre Rapsat, folder de son dernier album)
  <HR>
  <p>Le caddie virtuel sera calculé en : <jsp:getProperty name="thePreferences" property="uniteMonetaire" />
  </body>
</html>
```

Une exécution en local donne (le fond est bleu, je le jure) :



Bien sûr, c'est la couleur par défaut qui a été utilisée car le bean (dont la portée est d'ailleurs limitée à la page) a été instancié avec son constructeur par défaut. On peut le vérifier en ajoutant une trace dans ce constructeur (System.out.println("PreferencesWeb : constructeur par défaut");) : elle apparaît dans la console log de Tomcat intégré à Netbeans (que l'on peut obtenir en choisissant "View Server Log" dans le menu contextuel obtenu par un clic droit sur le nœud Servers- Tomcat dans l'onglet Runtime (5.5) ou Services (6)) :



Il serait évidemment plus logique que l'utilisateur choisisse sa couleur de fond, et même toutes ses préférences, au début afin que celles-ci soient prises en compte durant toute ses activités sur le site ...

9.4 Deux JSPs qui utilisent un bean session

Nous allons donc ajouter à la page de démarrage de l'application Web utilisée ci-dessus les champs de formulaires se référant à certaines préférences :

AccueilIdentification2.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">

<html>
  <head>
    <title></title>
  </head>
  <body>
    <H3>Bienvenue très cher(e) client(e) !!!!! </H3><p>
    <p>Veuillez vous identifier :</p>

    <form name="FormulaireIdentification" action="Identification2.jsp">
      <input type="text" name="nom" value="" size="50" /><p>
      <input type="text" name="prenom" value="" size="50" /><p>
      <input type="password" name="pwd" value="" size="30" /><p>
      Veuillez entrer :<p>
      Votre couleur de fond préférée :
      <select name="listeCouleurs">
        <option>blue</option>
        <option>red</option>
        <option>yellow</option>
        <option>white</option>
      </select></p>
      L'unité monétaire à utiliser :
      <input type="text" name="uniteMonetaire" value="EUR" /><p>
      <input type="submit" value="Ok" name="boutonOk" /><p>
    </form>

  </body>
</html>
```

Le JSP qui est sollicité est également un JSP déjà évoqué plus haut, mais qui a été légèrement modifié :

- ◆ la couleur choisie (champ listeCouleurs du formulaire) est récupérée et est mémorisée dans le bean Preferences : celui-ci est instancié avec le constructeur par défaut puis voit sa propriété couleurFond modifiée par un **jsp:setProperty**;
- ◆ le contrôle est transféré au JSP WelcomeUser.jsp (qui n'a (presque) pas été modifié) au moyen d'un simple lien HTML (on aurait pu imaginer mieux, mais bon ...);
- ◆ ceci ne fonctionnera que si le bean est déclaré dans la balise **jsp:useBean** comme étant de portée (scope) session (c'est page par défaut).

Identification2.jsp

```
<%@page language="java" %>
<%@page contentType="text/html; charset=ISO-8859-1"%>
<%@page pageEncoding="UTF-8"%>

<% @page info="(c) Claude Vilvens - 8/2007" %>

<% @page import="java.util.*" %>
<% @page import="java.text.*" %>

<jsp:useBean id="thePreferences" scope="session" class="Preferences.PreferencesWeb" />

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

<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>Réponse à l'identification et prise en compte des choix</title>
  </head>
  <body>

    <%! Date maintenant = new Date(); %>
    <%! String laDate = DateFormat.getDateInstance(DateFormat.FULL,
        DateFormat.FULL, Locale.FRANCE).format(maintenant); %>

    <!-- Page demandée le <%=laDate %> -->
    <h3>Bienvenue sur notre site très sérieux !</h3>
    Nous sommes le <%=laDate %> !!!
    <% String nom = request.getParameter("nom");
    String prenom = request.getParameter("prenom");
    String motDePasse = request.getParameter("pwd");
    %>

    <p>Nous vous saluons :
      <%= nom %> <%=prenom%> (**<%= motDePasse%> ***)<br>
    <p>(généré par <%=getServletInfo() %>)

    <jsp:setProperty name="thePreferences" property="couleurFond"
      value="<%="<%= request.getParameter("listeCouleurs")%>" />
```

```
</p> Couleur de fond qui sera utilisée : <jsp:getProperty name="thePreferences"  
property="couleurFond" /></p>  
<a href="WelcomeUser.jsp">La suite de l'histoire ...</a>  
  
</body>  
</html>
```

La seule modification dans le JSP WelcomeUser.jsp est de bien préciser que le bean préférences est de portée session :

WelcomeUser.jsp

```
<%@page language="java" %>  
<%@page contentType="text/html"%>  
<%@page pageEncoding="UTF-8"%>  
  
<jsp:useBean id="thePreferences" scope="session" class="Preferences.PreferencesWeb" />  
  
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"  
"http://www.w3.org/TR/html4/loose.dtd">  
  
<html>  
  <head>  
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">  
    <title>JSP Page with bean PreferencesWeb - Yeahh ... </title>  
  </head>  
  
<body bgcolor="<jsp:getProperty name="thePreferences" property="couleurFond" />">  
You know what ? I'm happy  
<p>(Pierre Rapsat, folder de son dernier album)  
<HR>  
<p>Le caddie virtuel sera calculé en : <jsp:getProperty name="thePreferences"  
property="uniteMonetaire" />  
  
</body>  
</html>
```

Un exemple d'exécution (depuis NetBeans pour utiliser le HTTP Monitor) est :

Mozilla Firefox

Fichier Édition Affichage Historique Marque-pages Outils ?

http://localhost:8084/IdentificationJsp/AccueilIdentification2.html Google

Hotmail Personnaliser les liens Windows Media Windows Le Soir en ligne - la_une

http://localhost:...ntification2.html

Bienvenue très cher(e) client(e) !!!!

Veuillez vous identifier :

Vilvens

Denys

xxxxxx

Veuillez entrer :

Votre couleur de fond préférée : yellow

L'unité monétaire à utiliser : EUR

Ok

Terminé

Réponse à l'identification et prise en compte des choix choisis - Mozilla Firefox

Fichier Édition Affichage Historique Marque-pages Outils ?

http://localhost:8084/IdentificationJsp/Identification2.jsp?nom=Vilvens&prenom=Deny Google

Hotmail Personnaliser les liens Windows Media Windows Le Soir en ligne - la_une

Réponse à l'identification et pris...

Bienvenue sur notre site très sérieux !

Nous sommes le lundi 20 août 2007 09 h 32 CEST !!!

Nous vous saluons : Vilvens Denys (**Pierre**)

(généré par (c) Claude Vilvens - 8/2007)

Couleur de fond qui sera utilisée : yellow [La suite de l'histoire ...](#)

Terminé

La suite de l'histoire ...'. A 'Terminé' button is at the bottom."/>

Usages Output Search Results HTTP Monitor

Request Cookies Session Context Client and Server Headers

Session
The session was created as a result of this request

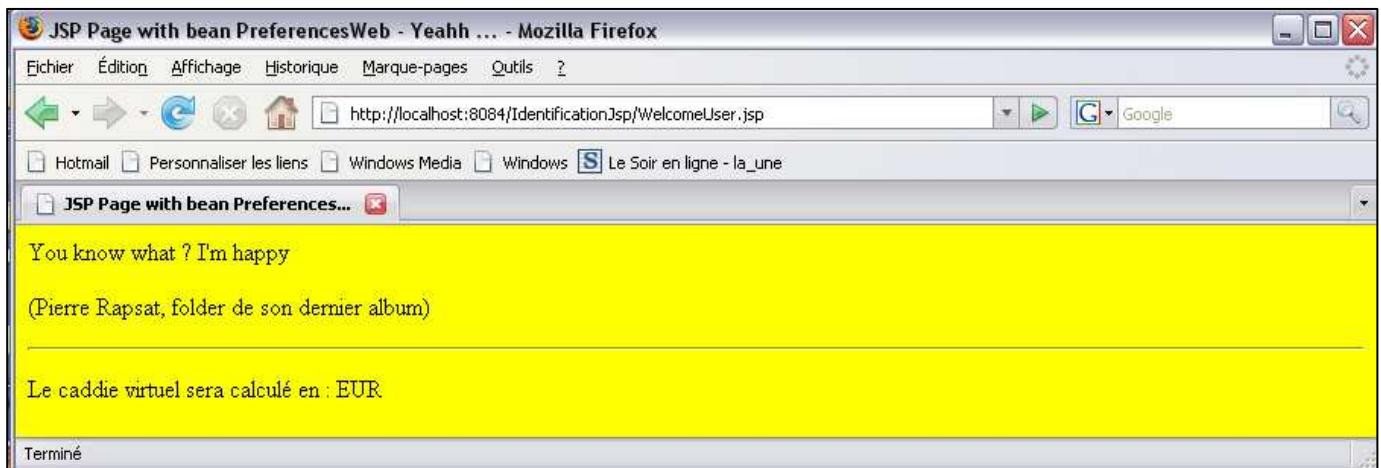
Session properties

Session ID	3B0EF84E4879A55BD3877717B563F931
Created	lundi 20 août 2007 9:32
Last accessed	lundi 20 août 2007 9:32
Max inactive interval	1800

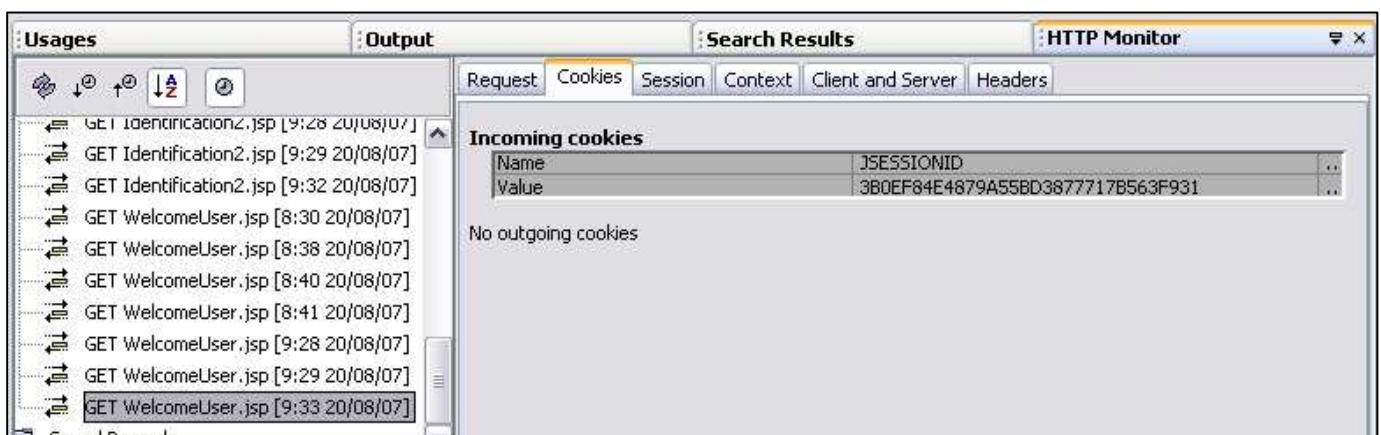
Session attributes after the request

thePreferences	Preferences.PreferencesWeb@1ce5e7a
----------------	------------------------------------

L'objet session contient donc bien un attribut "thePreferences" ! Finalement, on arrive par le lien sur (le fond est jaune cette fois, je le jure à nouveau) :



Le bean est toujours bien dans la session – et l'objet HttpSession fonctionne bien ici sur un cookie :



9.5 Un JSP et un bean avec Tomcat sous Linux

Le JSP et le bean développés ci-dessus en local fonctionnent sans problème sur un serveur Linux (ici, u2), à condition de placer le fichier class du bean dans le répertoire de son package (*Preferences*), ce dernier se trouvant dans le répertoire "classes" de WEB-INF :

```
-- java
`-- webdev
    |-- WEB-INF
    |   |-- classes
    |   |   `-- Preferences
    |   |       `-- PreferencesWeb.class
    |   |-- lib
    |   |-- jsp
    |   |   `-- jspuser.jsp
    |   |-- public_html
```

On obtient alors sans problème :



Si le bean est mémorisé dans un fichier jar, celui-ci sera placé dans le sous-répertoire "lib" de WEB-INF :

```
. 
|-- WEB-INF
|   |-- classes
|   `-- lib
|       `-- Preferences.jar
|-- jsp
|   `-- jspuser.jsp
`-- public_html
```

On peut encore placer l'appel du JSP dans une page HTML :

useBean.html

```
<!doctype html public "-//w3c//dtd html 4.0 transitional//en">
<html>
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
  <meta name="Author" content="Claude Vilvens">
  <meta name="GENERATOR" content="Mozilla/4.73 [en] (Windows NT 5.0; U) [Netscape]">
  <title>Utilisation d'un bean par un JSP</title>
</head>
<body>
<b><font color="#FF0000"><font size=+3>Utilisation
d'un bean par un JSP</font></font></b>
<br>&nbsp;

<form method="post" action="http://u2:8180/webdev2/jsp/jspuser.jsp">
<P><input type="submit" value="à l'assaut"></P>
</form>

</body>
</html>
```

La page en question est placée où bon nous semble, par exemple dans un répertoire public_html :

```
.
|-- WEB-INF
|   |-- classes
|   |   `-- Preferences
|   |       `-- PreferencesWeb.class
|   `-- lib
|-- jsp
|   `-- jspuser.jsp
`-- public_html
    `-- useBean.html
```

10. L'accès aux bases de données par les JSP au moyen des Java Beans

La séparation entre la présentation et la logique de programmation est le fondement même des JSP. Si un JSP désire réaliser un accès à une base de données, il peut bien sûr comporter directement les instructions JDBC réalisant l'accès; mais le code JSP correspondant est alors un mélange de Java et de balises de présentation ...

10.1 Le modèle à 1 bean

Pour éviter cela, **le JSP peut utiliser les services d'un bean responsable de la connexion à la base données et de l'exécution des requêtes**. Celui-ci comportera des variables membres qui sont un objet Connection et un objet Statement. En effet, ceux-ci doivent conserver leur valeur durant toute l'existence du bean. On peut aussi prévoir comme variables membres les nom d'utilisateur et mot de passe, pour les SGBD qui connaissent ces notions. La requête à exécuter peut également être **une propriété configurable** du bean afin

de pouvoir préciser la requête à exécuter dans le JSP – mais on peut aussi bien préciser cette requête comme ***paramètre de la méthode*** réalisant la sélection; les deux versions seront donc prévues. Si beanAccess est le bean en question, il peut s'écrire (dans un package beansForJdbc) :

```
beanAccess.java
/*
 * beanAccess.java
 */
package beansForJdbc;

import java.beans.*;
import java.sql.*;

/**
 * @author Vilvens
 */
public class beanAccess extends Object implements java.io.Serializable
{
    /** Variable membre de la propriété DataBase.
     */
    public String dataBase;
    /** Variable membre de la propriété PiloteJdbc.
     */
    public String driverJdbc;

    /** Variable membre.
     * Représente la connexion en cours.
     */
    private Connection con;
    /** Variable membre.
     * Représente l'instruction en cours.
     */
    private Statement instruc;

    /** Holds value of property user. */
    private String user;
    /** Holds value of property password. */
    private String password;

    /** Holds value of property requete. */
    private String requete;

    /** Creates new beanAccess */
    public beanAccess()
    {
        driverJdbc = "sun.jdbc.odbc.JdbcOdbcDriver";
        user = ""; password = "";
    }
}
```

```
/** Creates new beanAccess.  
 * @param db New value of property DataBase.  
 * @param dv New value of property piloteJdbc.  
 */  
public beanAccess (String dv, String db)  
{  
    driverJdbc = dv;  
    DataBase = db;  
    user = ""; password = "";  
}  
  
/** Creates new beanAccess.  
 * @param db New value of property DataBase.  
 * @param dv New value of property piloteJdbc.  
 * @param u New value of property user.  
 * @param p New value of property password.  
 */  
public beanAccess (String dv, String db, String u, String p)  
{  
    driverJdbc = dv;  
    DataBase = db;  
    user = u;  
    password = p;  
}  
  
/** Getter for property DataBase.  
 * @return Value of property DataBase.  
 */  
public String DataBase() { return DataBase; }  
/** Setter for property DataBase.  
 * @param DataBase New value of property DataBase.  
 */  
public void setDataBase(String DataBase)  
{  
    this.DataBase = DataBase;  
}  
/** Getter for property piloteJdbc.  
 * @return Value of property piloteJdbc.  
 */  
public String getPiloteJdbc() { return driverJdbc; }  
/** Setter for property piloteJdbc.  
 * @param piloteJdbc New value of property piloteJdbc.  
 */  
public void setPiloteJdbc(String piloteJdbc)  
{  
    driverJdbc = piloteJdbc;  
}  
/** Getter for property password.  
 * @return Value of property password.  
 */
```

```

public String getPassword() { return password; }
/** Setter for property password.
 * @param password New value of property password.
 */
public void setPassword(String password)
{
    this.password = password;
}

/** Getter for property user.
 * @return Value of property user.
 */
public String getUser() { return user; }
/** Setter for property user.
 * @param user New value of property user.
 */
public void setUser(String user)
{
    this.user = user;
}

/** Méthode de connexion à une base de données.
 */
public void creerConnexionBD() throws Exception
{
    Class.forName(getPiloteJdbc());
    con = DriverManager.getConnection(getDataBase(),getUser(),getPassword());
    instruc = con.createStatement();
}

/** Méthode d'exécution d'une requête de sélection.
 * La requête sous forme de chaîne de caractères est la variable membre
 */
synchronized public ResultSet executeRequeteSelection() throws Exception
{
    return instruc.executeQuery(getRequete()); // synchronized() needed ???
}

/** Méthode d'exécution d'une requête de sélection.
 * @param r La requête sous forme de chaîne de caractères
 */
synchronized public ResultSet executeRequeteSelection(String r) throws Exception
{
    return instruc.executeQuery(r);
}

/** Getter for property requete.
 * @return Value of property requete.
 */

```

```

public String getRequete() { return requete; }
/** Setter for property requete.
 * @param requete New value of property requete.
 */
public void setRequete(String r)
{
    requete = r;
}

```

Notre JSP peut alors s'écrire, en considérant la requête comme une propriété du bean :

SelectProduitsMarie.jsp (version à un bean)

```

<%@page contentType="text/html"%>
<%@page import="java.sql.*" %>

<jsp:useBean id="connectJdbcBean" class="beansForJdbc.beanAccess" >
<jsp:setProperty name="connectJdbcBean" property="piloteJdbc"
value="sun.jdbc.odbc.JdbcOdbcDriver" />
<jsp:setProperty name="connectJdbcBean" property="dataBase" value="jdbc:odbc:marie" />
<jsp:setProperty name="connectJdbcBean" property="requete" value="SELECT * FROM
PRODUITS" />
</jsp:useBean>

<html>
<head><title>JSP + Java Beans = la classe</title></head>
<body>
<H3>MARIE-INPRES !!! Les bons légumes de chez nous ;-)</H3><BR>

<% connectJdbcBean.creerConnexionBD();
ResultSet rs = connectJdbcBean.executeRequeteSelection();
while (rs.next())
{
%
Produit : <%= rs.getString("codePr") %><BR>
Descriptif : <%= rs.getString("descriptif") %><BR>
Propriétés : <%= rs.getString("proprietes") %><BR>
<%= chaineTaille (rs.getFloat("taille1")) %><BR>
<HR>

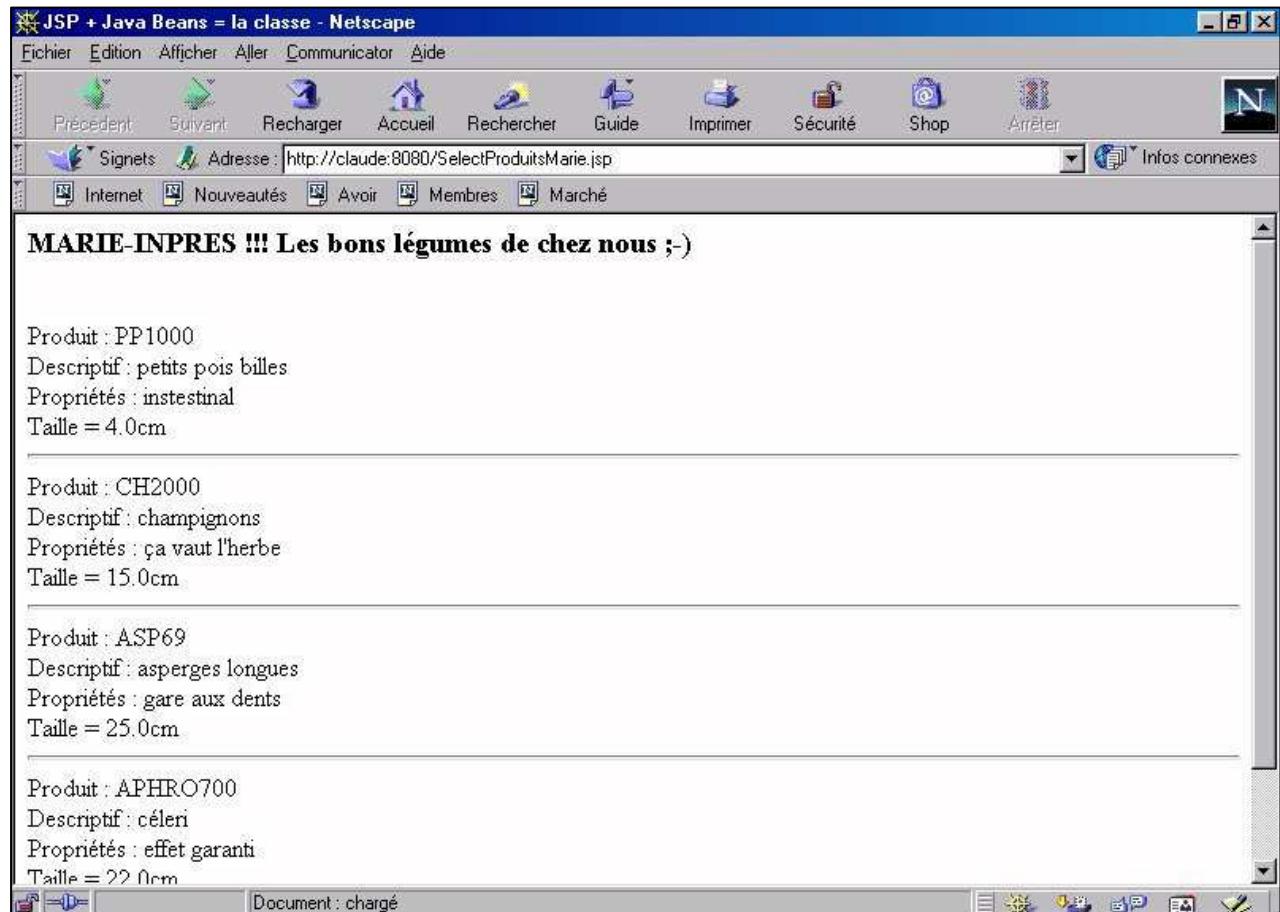
<%
}
%>

<% ! public String chaineTaille(float taille)
{
    return "Taille = " + new Float(taille).toString() + "cm";
}
%>

```

```
<BR>
Et voilà le travail ...
</body>
</html>
```

Un résultat :



10.2 Le modèle à 2 beans

On peut obtenir le même résultat en utilisant un deuxième bean chargé de représenter un tuple fourni comme réponse à la requête. L'intérêt est de placer le résultat d'une requête directement dans des éléments manipulables par des utilisateurs non programmeurs (= le credo des JSP)⁸. En effet, de cette manière, le code Java est parfaitement séparé des tags de présentation. Ce bean beanMarieProduits s'écrit :

beanMarieProduits.java

```
/*
 * beanMarieProduits.java
 */
package beansForJdbc;

import java.beans.*;
```

⁸ clin d'œil : le bean d'accès la base de données est ce que l'on appellera un bean session dans le contexte des EJB, tandis que le bean tuple est une espèce de bean entité ;-) ...

```

/**
 * @author Vilvens
 */
public class beanMarieProduits extends Object implements java.io.Serializable
{
    /** Holds value of property codeProduit. */
    private String codeProduit;
    /** Holds value of property descriptif. */
    private String descriptif;
    /** Holds value of property proprietes. */
    private String proprietes;
    /** Holds value of property taille. */
    private float taille;

    /** Creates new beanMarieProduits */
    public beanMarieProduits() { }

    /** Getter for property codeProduit.
     * @return Value of property codeProduit.
     */
    public String getCodeProduit() { return codeProduit; }
    /** Setter for property codeProduit.
     * @param codeProduit New value of property codeProduit.
     */
    public void setCodeProduit(String codeProduit) { this.codeProduit = codeProduit; }

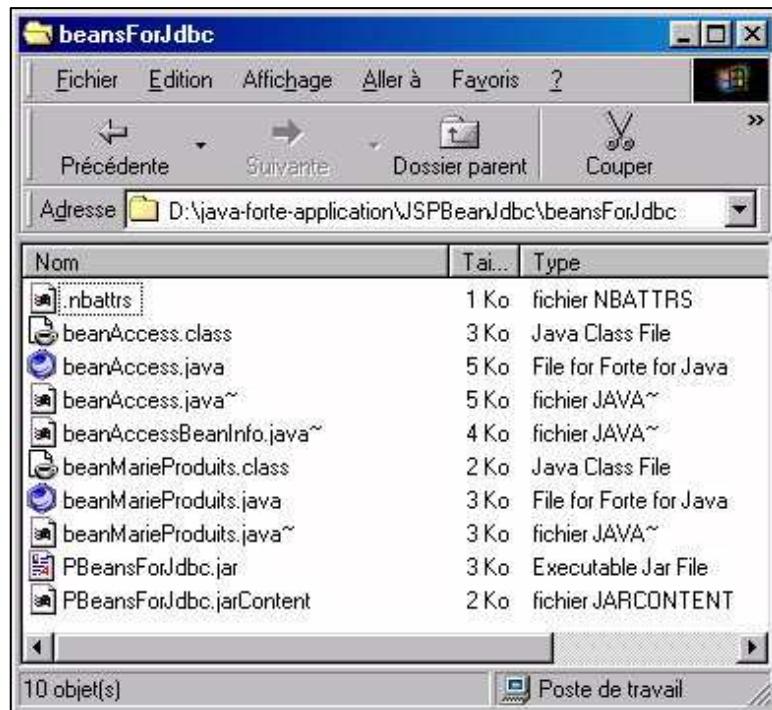
    /** Getter for property descriptif.
     * @return Value of property descriptif.
     */
    public String getDescriptif() { return descriptif; }
    /** Setter for property descriptif.
     * @param descriptif New value of property descriptif.
     */
    public void setDescriptif(String descriptif) { this.descriptif = descriptif; }

    /** Getter for property proprietes.
     * @return Value of property proprietes.
     */
    public String getProprietes() { return proprietes; }
    /** Setter for property proprietes.
     * @param proprietes New value of property proprietes.
     */
    public void setProprietes(String proprietes) { this.proprietes = proprietes; }

    /** Getter for property taille.
     * @return Value of property taille.
     */
    public float getTaille() { return taille; }
    /** Setter for property taille.
     * @param taille New value of property taille.
     */
    public void setTaille(float taille) { this.taille = taille; }
}

```

Ici, plutôt que de travailler en war (leur usage n'est pas obligatoire, rappelons-le), les deux beans sont placés dans le même package beansForJdbc puis dans un fichier jar PBeansForJdbc.jar :



Notre JSP revu en tenant compte de l'utilisation du deuxième bean s'écrit :

SelectProduitsMarie.jsp (version à deux beans)

```
<%@page contentType="text/html"%>
<%@page import="java.sql.*" %>

<jsp:useBean id="tupleBean" class="beansForJdbc.beanMarieProduits" />

<jsp:useBean id="connectJdbcBean" class="beansForJdbc.beanAccess" >
<jsp:setProperty name="connectJdbcBean" property="piloteJdbc"
value="sun.jdbc.odbc.JdbcOdbcDriver" />
<jsp:setProperty name="connectJdbcBean" property="dataBase" value="jdbc:odbc:marie" />
<jsp:setProperty name="connectJdbcBean" property="requete" value="SELECT * FROM
PRODUITS" />
</jsp:useBean>

<html>
<head><title>JSP + Java Beans = la classe</title></head>
<body>
<H3>MARIE-INPRES !!! Les bons légumes de chez nous ;-) </H3><BR>

<% connectJdbcBean.creerConnexionBD();
ResultSet rs = connectJdbcBean.executeRequeteSelection();
```

```

while (rs.next())
{
%>
    <jsp:setProperty name="tupleBean" property="codeProduit" value='<%= rs.getString("codePr")'
%>' />
    <jsp:setProperty name="tupleBean" property="descriptif" value='<%= rs.getString("descriptif")'
%>' />
    <jsp:setProperty name="tupleBean" property="proprietes" value='<%= rs.getString("proprietes")'
%>' />
    <jsp:setProperty name="tupleBean" property="taille" value='<%=rs.getFloat("taille1") %>' />

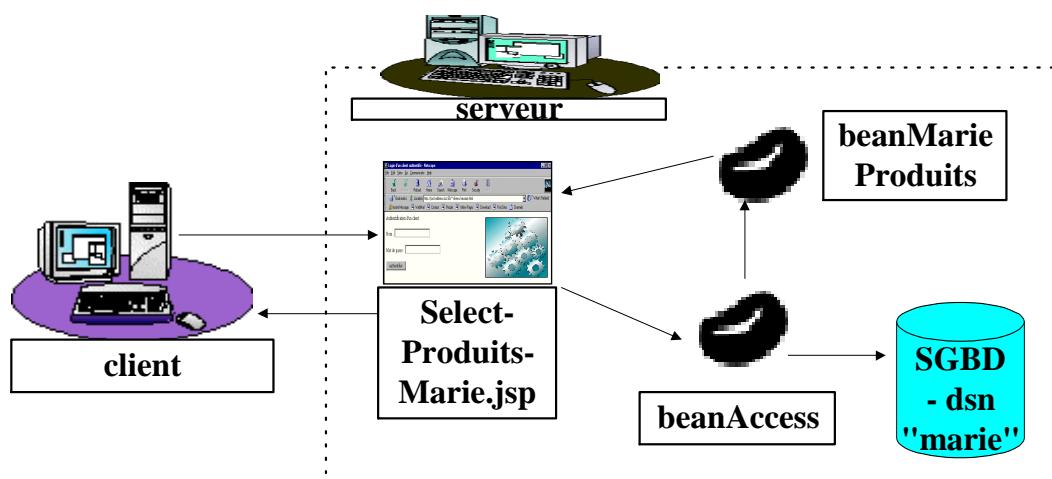
    Produit : <jsp:getProperty name="tupleBean" property="codeProduit" /><BR>
    Descriptif : <jsp:getProperty name="tupleBean" property="descriptif" /><BR>
    Propriétés : <jsp:getProperty name="tupleBean" property="proprietes" /><BR>
    Taille : <jsp:getProperty name="tupleBean" property="taille" /><BR>
<HR>
<%
}
%>

<%! public String chaineTaille(float taille)
{
    return "Taille = " + new Float(taille).toString() + "cm";
}
%>

<BR>
Et voilà à nouveau le travail ...
</body>
</html>

```

Le résultat sur l'écran du client est exactement le même. Donc, en résumé :



Remarque

Dans le cas d'un Tomcat serveur Web standalone, un déploiement sans war et sans jar demandera la disposition dans les répertoires suivante :

```
.  
|-- WEB-INF  
| | |-- classes  
| | | `-- beansForJdbc  
| | |   |-- beanMarieProduits.class  
| | |   |-- beanAccess.class  
| | `-- lib  
|-- JSP  
  '-- SelectProduitsMarie.jsp
```

alors qu'avec un jar contenant le package en question (ici PackageBeansForJdbc.jar), on aura :

```
.  
|-- WEB-INF  
| | |-- classes  
| | | `-- lib  
| | |   '-- PackageBeansForJdbc.jar  
|-- JSP  
  '-- SelectProduitsMarie.jsp
```

11. Les pages d'erreurs

Jusqu'à présent, nous ne nous sommes guère tracassés des erreurs éventuelles – au point de gérer les exceptions de la manière la plus simple qui soit, c'est-à-dire de ne pas les gérer du tout ☺ ...

Résultat : l'utilisateur peut voir arriver des choses merveilleuses en cas d'exception, du genre l'affichage de la pile des appels (l'effet du printStackTrace()). Il est possible de lui éviter cet affichage agressif **en définissant soi-même la page à laquelle le contrôle sera transféré en cas d'erreur**. Cette page est en fait un JSP, caractérisé par un attribut **isErrorHandler** de la directive @page :

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

Le JSP correspondant se définit de la même manière qu'un JSP normal :

ExceptionMarie.jsp

```
<%@page contentType="text/html"%>  
<%@page isErrorPage="true" %>  
<html>  
<head><title>JSP Page d'erreur</title></head>  
<body>  
<H3>Oh oh ... Une lamentable erreur vient de se produire :-( ...</H3>  
<BR><H2><%=exception %></h2>  
</body>  
</html>
```

On remarquera l'utilisation de l'objet implicite exception, qui n'est disponible précisément que dans les pages d'erreur pour lesquelles isErrorPage est true.

L'attribut errorPage de la directive @page permet à un JSP de spécifier à quelle page d'erreur il faut faire appel en cas de problème :

```
<%@page errorPage="<nom du JSP d'erreur>"%>
```

Reprendons notre JSP utilisant deux beans pour accéder à la source de données "marie". Remplaçons le nom de cette source par "petiteMarie", source qui n'existe pas : une exception sera donc lancée. Mais nous prévoyons que, dans ce cas, le contrôle sera passé au JSP ExceptionMarie.jsp :

SelectProduitsMarie.jsp (version avec erreur)

```
<%@page contentType="text/html"%>
<%@page errorPage="ExceptionMarie.jsp"%>
<%@page import="java.sql.*" %>

<jsp:useBean id="tupleBean" class="beansForJdbc.beanMarieProduits" />
<jsp:useBean id="connectJdbcBean" class="beansForJdbc.beanAccess" >
<jsp:setProperty name="connectJdbcBean" property="piloteJdbc"
value="sun.jdbc.odbc.JdbcOdbcDriver" />
<jsp:setProperty name="connectJdbcBean" property="dataBase"
value="jdbc:odbc:petiteMarie" />
<jsp:setProperty name="connectJdbcBean" property="requete" value="SELECT * FROM
PRODUITS" />
</jsp:useBean>

<html>
<head><title>JSP + Java Beans = la classe</title></head> ...
</html>
```

Effectivement, la réponse ne se fait pas attendre :



Bien sûr, en pratique, l'utilisateur final, probablement un client end-user, ne sera guère intéressé par le message obtenu. En fait, on peut imaginer de lui afficher un message rassurant tandis que le diagnostic d'erreur précis est transmis aux gestionnaires du serveur :

- ◆ par écriture dans un fichier log qui sera lu périodiquement par ces gestionnaires;
- ◆ par envoi d'un e-mail évocateur au gestionnaire concerné.

12. Les balises personnalisées

12.1 Une classe Java pour une balise sans corps

Toujours pour accroître la séparation entre présentation et logique, les JSP permettent la définition de nouvelles balises d'action qui seront rassemblées et définies dans des **bibliothèques de balises**. *Chaque balise personnalisée est implémentée au moyen d'une classe Java*. Une telle bibliothèque est décrite par un **TLD** (Tag Library Descriptor) – concrètement, *un fichier XML d'extension tld*. On s'en doute (puisque l'on prend goût à la simplicité), il suffit de spécifier que l'on va faire référence à une bibliothèque bien précise pour pouvoir ensuite utiliser toutes les nouvelles balises correspondantes. La déclaration de l'utilisation d'une balise se fait au moyen de la directive :

```
<%@ taglib uri=<fichier tld> prefix=<préfixe de tous les tags de cette bibliothèque>%>
```

Concrètement, quel est l'intérêt pratique de ces possibilités ? Toujours la même réponse : séparer au maximum la présentation du JSP, accessible aux non-programmeurs, de la logique interne, réservée aux développeur(se)s tatoué(e)s. Pour illustrer cela, nous allons développer **une balise personnalisée sans corps**, soit une balise qui permettra par exemple d'insérer la date courante. Son utilisation concrète devrait ressembler à ceci :

```
<outil:maintenant langue="IT" />
```

ce qui aura pour effet d'insérer, au moyen du tag "maintenant" de la balise "outil" (ce préfixe "outil" est défini dans la directive taglib ci-dessus), la date courante exprimée dans la langue précisée par l'attribut "langue" (ici, l'italien).

12.2 La classe balise

Nous allons tout d'abord écrire une classe, appelons-la BaliseDateDuJour, dont le rôle sera d'encapsuler la logique de notre balise : on parle encore de "**tag handler**" pour une telle classe. Une telle classe doit implémenter l'interface **Tag**, défini dans le package javax.servlet.jsp.tagext; cet interface déclare 6 méthodes. Mais il existe aussi une classe **TagSupport** du même package qui fournit une implémentation par défaut de l'interface. Dans notre cas, nous dériverons notre classe balise de cette en ne redéfinissant que les méthodes pour lesquelles c'est nécessaire, soit ici :

- ◆ public int **doStartTag()** throws JspException

Méthode appelée dès que le moteur à JSP rencontre la balise d'ouverture – elle contient donc clairement les initialisations nécessaires; la valeur renournée permet de signifier si le corps de la balise doit être ignoré (la valeur est la constante de classe de Tag **SKIP_BODY** – ce sera le cas ici) ou au contraire être traité (la valeur est la constante de classe de Tag **EVAL_BODY_INCLUDE**).

- ◆ public int **doEndTag()** throws JspException

Méthode appelée dès que le moteur à JSP rencontre la balise de fermeture; c'est dans cette méthode que la tâche dévolue à la balise est effectivement programmée; la valeur renournée permet de signifier si le reste de la page doit être traité (la valeur est la constante de classe de Tag **EVAL_PAGE** – ce sera le cas ici) ou si ce traitement doit être interrompu (la valeur est la constante de classe de Tag **SKIP_PAGE**).

Notre classe devrait donc avoir l'aspect suivant (même si, en pratique, nous nous servirons du code généré par NetBeans) :

BaliseDateDuJour.java

```
/*
 * BaliseDateDuJour.java
 */
/** 
 * @author Vilvens
 */

import javax.servlet.jsp.tagext.*;
import javax.servlet.jsp.*;
import java.io.*;
import java.util.*;
import java.text.*;

public class BaliseDateDuJour extends TagSupport
{
    private String chDate;
    private String langue;

    public BaliseDateDuJour() { langue="FR"; }

    String getLangue () { return langue; }
    public void setLangue (String l) { langue = l; }

    public int doStartTag()
    {
        try
        {
            pageContext.getOut().println("(selon le serveur)");
            // pour dire quelque chose dans la page réponse au client ...
        }
        catch (IOException e) {}
        return SKIP_BODY;
    }

    public int doEndTag()
    {
        Date maintenant = new Date();
        if (langue == "UK")
            chDate = DateFormat.
                getTimeInstance(DateFormat.FULL,DateFormat.FULL,Locale.UK).
                format(maintenant);
        else
            if (langue == "IT")
                chDate = DateFormat.
                    getTimeInstance(DateFormat.FULL,DateFormat.FULL,Locale.ITALY).
                    format(maintenant);
            else
                if (langue == "FR")
```

```

chDate = DateFormat.
    getTimeInstance(DateFormat.FULL,DateFormat.FULL,Locale.FRANCE).
    format(maintenant);
else
chDate = DateFormat.
    getTimeInstance(DateFormat.FULL,DateFormat.FULL,Locale.US).
    format(maintenant);

try
{
    pageContext.getOut().println("*** " + chDate + " *** (" + getLangue() + ") ***");
}
catch (IOException e) {}
return EVAL_PAGE;
}
}

```

Pour écrire dans la page réponse au client, on a utilisé ici la variable membre protected de TagSupport nomme *pageContext* : celle-ci, instance de **PageContext** (package javax.servlet.jsp) donne accès à tous les attributs et les espaces de noms associés au JSP. En particulier, la méthode :

public abstract JspWriter **getOut()**

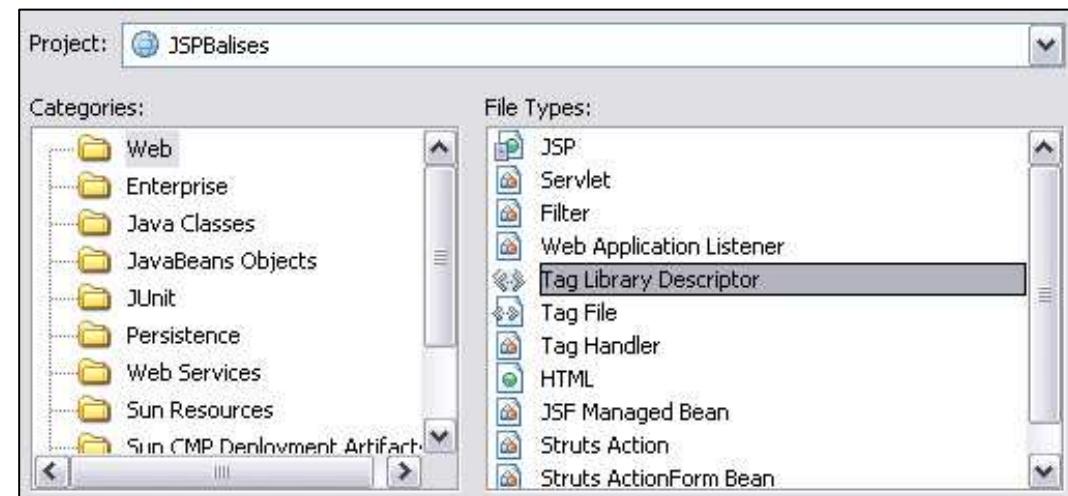
fournit un flux qui implémente la classe abstraite **JspWriter** et qui permet d'"écrire" dans le JSP (il s'agit d'une espèce de PrintWriter mais orienté JSP et connaissant les exceptions).

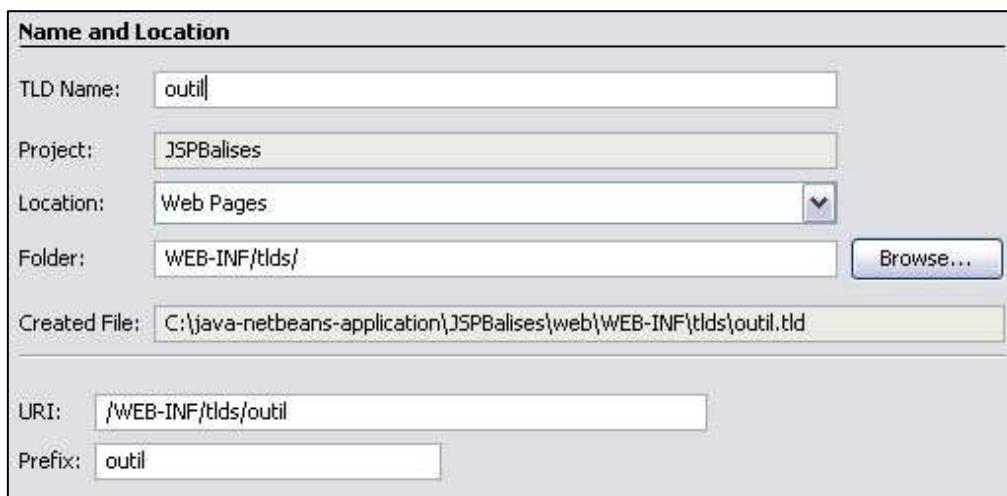
Remarque

L'interface **BodyTag** (dérivé de Tag) et la classe **BodyTagSupport** (dérivée de TagSupport) sont les versions plus complexes à utiliser pour les balises personnalisées qui utilisent leur corps (beek ;-) ...) – nous y reviendrons.

12.3 La création d'un tld

Nous allons à présent définir le TLD depuis NetBeans. Nous sommes supposés travailler dans une application Web et le TLD sera sensé se trouver dans un sous-répertoire **tlds** de WEB-INF. Dans le répertoire WEB-INF, on ajoute un objet "Tag Library Descriptor", dont le nom sera donc, d'après ce que nous avons prévu, "outil" :





Le fichier tld de base ressemble à ceci :

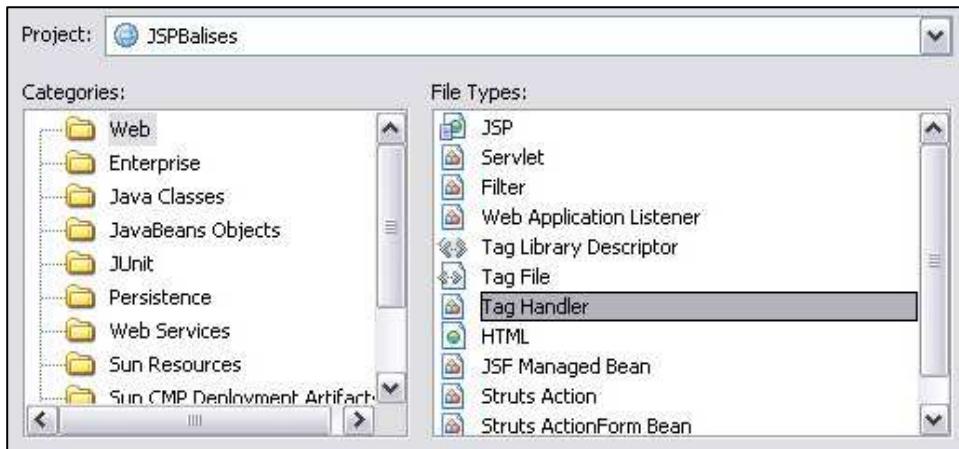
```
outil.tld (version de base)
<?xml version="1.0" encoding="UTF-8"?>

<taglib version="2.0" xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee web-jsptaglibrary_2_0.xsd">
<tlib-version>1.0</tlib-version>
<short-name>outil</short-name>
<uri>/WEB-INF/tlds/outil</uri>
<!-- A validator verifies that the tags are used correctly at JSP
     translation time. Validator entries look like this:>
<validator>
    <validator-class>com.mycompany.TagLibValidator</validator-class>
    <init-param>
        <param-name>parameter</param-name>
        <param-value>value</param-value>
    </init-param>
</validator>
-->
<!-- A tag library can register Servlet Context event listeners in
     case it needs to react to such events. Listener entries look
     like this:>
<listener>
    <listener-class>com.mycompany.TagLibListener</listener-class>
</listener>
-->
</taglib>
```

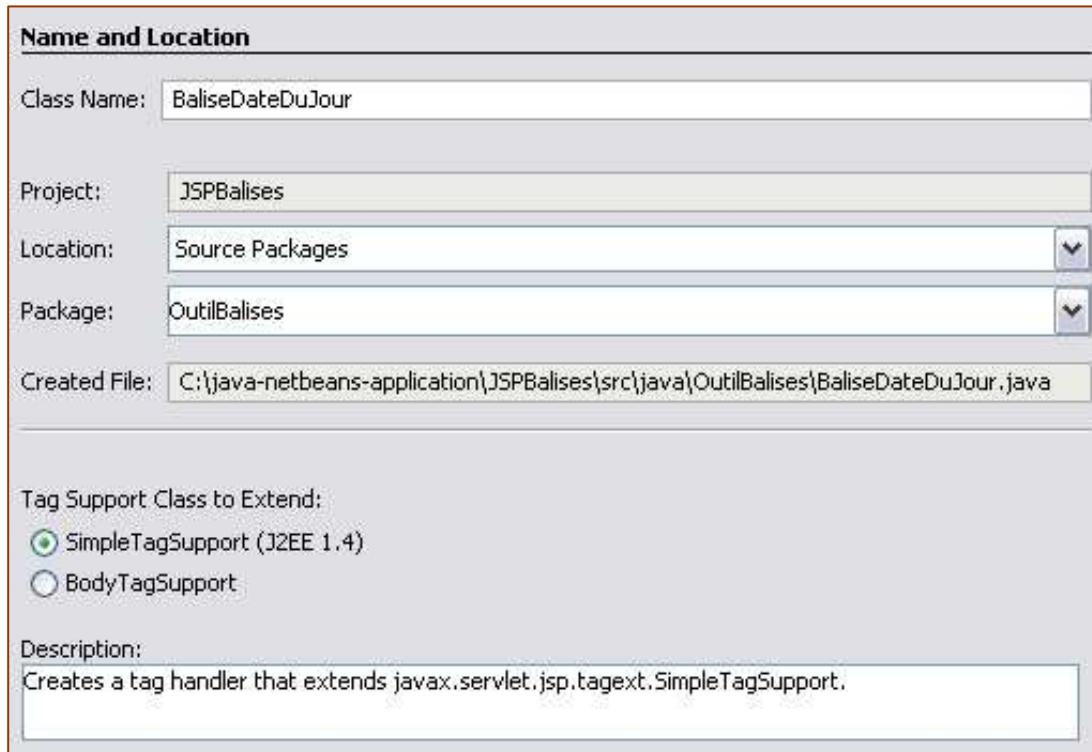
Le résultat brut de la génération a effectivement créé le "**shortname**". Ce shortname est celui qui servira de préfixe dans l'utilisation future de notre balise personnalisée. Ce fichier sera complété au fur et à mesure que nous développerons des balises nouvelles.

12.4 La création du tag handler

Pour rappel, il s'agit de la classe qui contient la logique associée à notre balise. Nous créons donc cette classe :

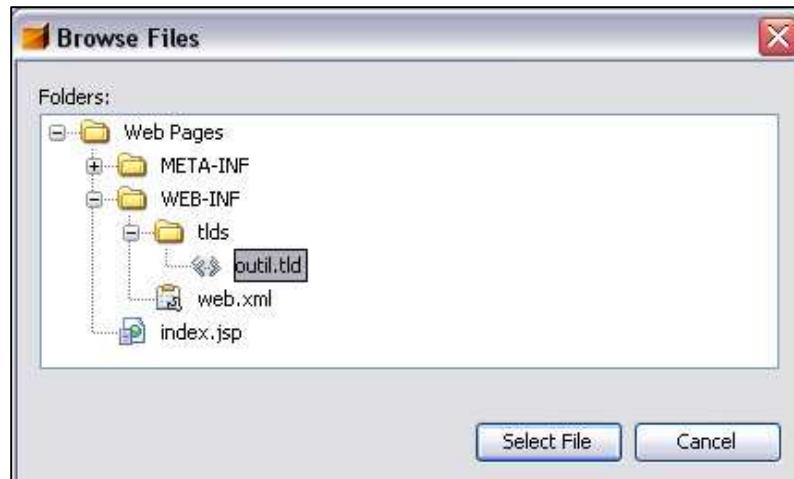


Nous en fournissons alors le nom (BaliseDateDuJour), nous la plaçons dans un package (OutilBalises) et nous précisons qu'elle dérive de SimpleTagSupport :

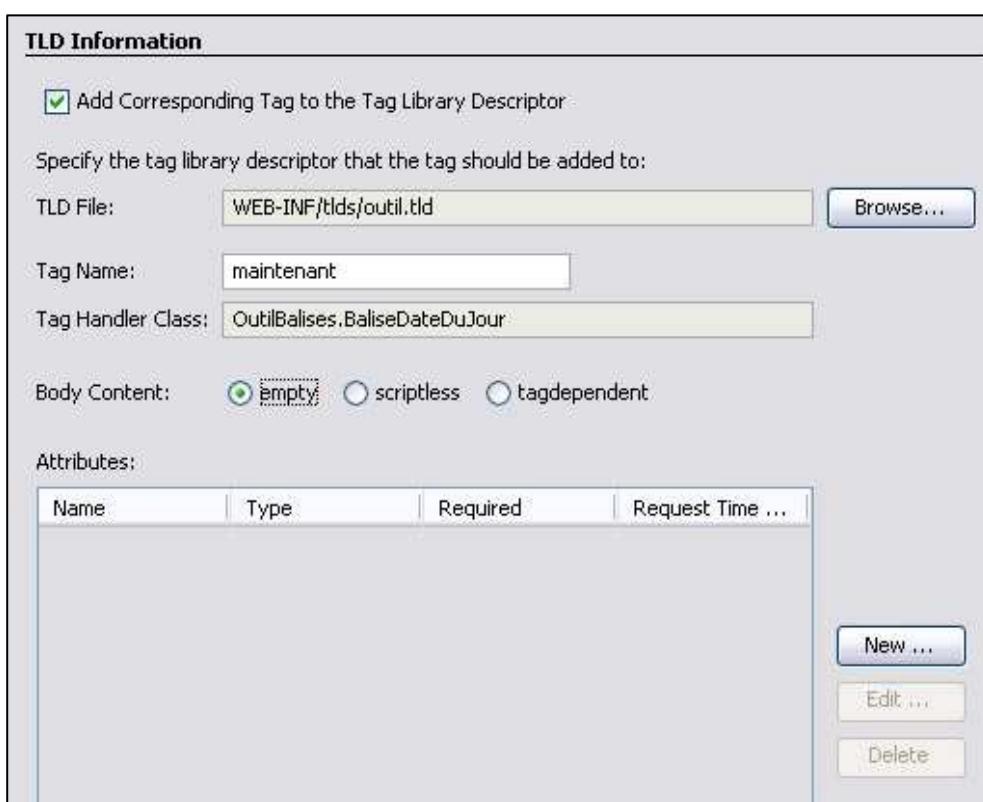


Cette classe **SimpleTagSupport** implémente l'interface **SimpleTag**, tous du package `javax.servlet.jsp.tagext`. Cet interface `SimpleTag` ressemble à l'interface `Tag` évoqué plus haut. Cependant, au lieu d'utiliser les méthodes `doStartTag()` et `doEndTag()`, il est pensé selon une méthode `doTag()` plus globale qui est invoquée à chaque utilisation de la balise.

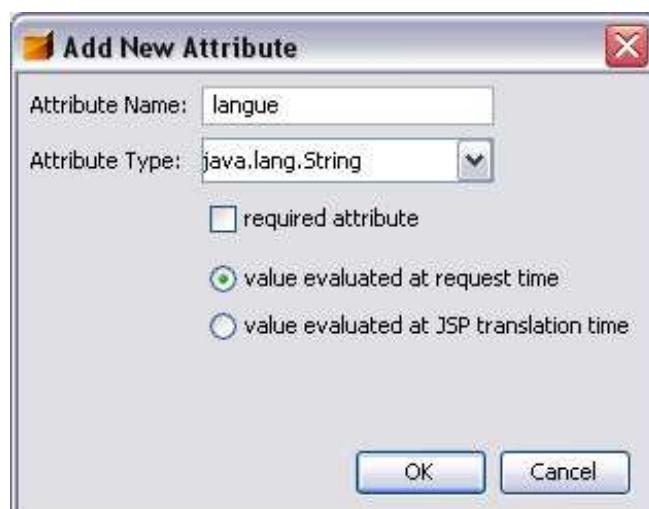
Dans le panneau suivant du wizzard, il nous faudra préciser dans quel tld la relation entre la classe et la balise doit être consignée :



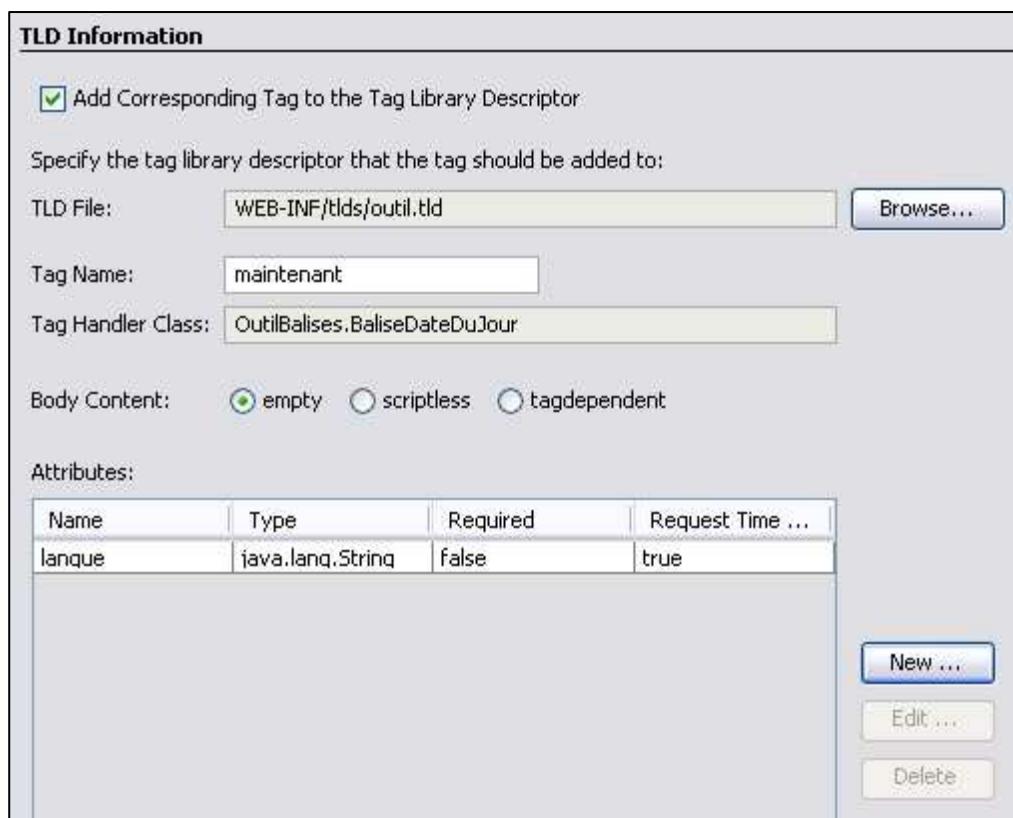
et bien sûr préciser le nom du tag qui sera utilisé :



Nous pouvons à présent définir l'attribut langue :



pour en arriver à ceci :



Le fichier tld a été modifié en conséquences de ces définitions :

```
outil.tld (version modifiée)
<?xml version="1.0" encoding="UTF-8"?>
<taglib version="2.0" xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee web-jsptaglibrary_2_0.xsd">
<tlib-version>1.0</tlib-version>
<short-name>outil</short-name>
<uri>/WEB-INF/tlds/outil</uri>
...
<tag>
<name>maintenant</name>
<tag-class>OutilBalises.BaliseDateDuJour</tag-class>
<body-content>empty</body-content>
<attribute>
<name>langue</name>
<rtpexprvalue>true</rtpexprvalue>
<type>java.lang.String</type>
</attribute>
</tag>
</taglib>
```

Nous obtenons ainsi une classe handler de base, dans laquelle nous ajoutons les actions souhaitées :

BaliseDateDuJour.java (avec génération de la base par netBeans)

```
/*
 * BaliseDateDuJour.java
 */

package OutilBalises;

import javax.servlet.jsp.tagext.*;
import javax.servlet.jsp.JspWriter;
import javax.servlet.jsp.JspException;
import java.util.*;
import java.text.*;

/**
 * @author Vilvens
 */

public class BaliseDateDuJour extends SimpleTagSupport
{
    private String chDate;
    private java.lang.String langue;
    public void doTag() throws JspException
    {
        JspWriter out=getJspContext().getOut();

        try
        {
            // TODO: insert code to write html before writing the body content.
            // e.g.:
            //
            // out.println("<strong>" + attribute_1 + "</strong>");
            // out.println("  <blockquote>");
            out.println("(selon le serveur)");

            JspFragment f=getJspBody();
            if (f != null) f.invoke(out);

            // TODO: insert code to write html after writing the body content.
            // e.g.:
            //
            // out.println("  </blockquote>");
            Date maintenant = new Date();
            if (langue == "UK")
                chDate = DateFormat.getDateInstance(DateFormat.FULL,
                    DateFormat.FULL, Locale.UK).format(maintenant);
            else
        }
    }
}
```

```

if (langue == "IT")
    chDate = DateFormat.getDateInstance(DateFormat.FULL,
        DateFormat.FULL, Locale.ITALY).format(maintenant);
else
if (langue == "FR")
    chDate = DateFormat.getDateInstance(DateFormat.FULL,
        DateFormat.FULL, Locale.FRANCE).format(maintenant);
else chDate = DateFormat.getDateInstance(DateFormat.FULL,
    DateFormat.FULL, Locale.US).format(maintenant);

    out.println("*** " + chDate + " *** (" + getLangue() + ") ***");
}
catch (java.io.IOException ex)
{
    throw new JspException(ex.getMessage());
}

/***
 * Setter for the langue attribute.
 */
public void setLangue(java.lang.String value)
{
    this.langue = value;
}
/***
 * Getter for property langue.
 * @return Value of property langue.
 */
public java.lang.String getLangue()
{
    return langue;
}
}

```

12.5 La classe SimpleTagSupport

La méthode doTag() rassemble donc l'équivalent de l'appel des méthodes doStartTag() et doEndTag() de la classe TagSupport, ne tenant à priori pas compte du corps de la balise. Cependant, si ce corps doit être manipulé, les méthodes :

```

protected JspFragment getJspBody()
public void setJspBody(JspFragment jspBody)

```

le permettent – la classe qui implémentera la classe abstraite JspFragment est chargée d'encapsuler ce corps.

Il faut encore savoir que toute classe dérivée de SimpleTagSupport doit avoir un constructeur par défaut (explicite ou hérité implicitement). Le cycle de vie d'un tel handler lorsque la balise sera rencontrée est celui-ci :

- ◆ appel du constructeur par défaut par le container Web;
- ◆ cet appel sera suivi par l'appel de `setJspContext()` ...
- ◆ et des méthodes `setXXX()` pour les attributs;
- ◆ si il y a un corps (ce qui ne sera pas le cas ci), c'est la méthode `setJspBody()` qui est appelée;
- ◆ enfin, la méthode `doTag()` est invoquée.

12.6 Le JSP utilisateur de la balise

Il nous reste à présent à créer un JSP qui fera usage de notre balise "maintenant", appartenant à notre bibliothèque préfixée par "outil" et qui fera éventuellement usage de son attribut "langue".

WelcomeTagDate.jsp

```
<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>
<%-- 
<%@taglib uri="/WEB-INF/tlds/outil.tld" prefix="outil" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
 "http://www.w3.org/TR/html4/loose.dtd">

<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>JSP utilisant une balise personnalisée</title>
  </head>
  <body>

    <h1>Bienvenue :-) </h1>
Et bonjour ! Nous sommes le :

<outil:maintenant langue="IT" />
  </body>
</html>
```

L'exécution directe d'un tel jsp en local sous NetBeans donne :



tandis que l'affichage du code source dans le browser donne simplement :

Page HTML générée par le JSP

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

<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>JSP utilisant une balise personnalisée</title>
  </head>
  <body>
    <h1>Bienvenue :-)</h1>
    Et bonjour ! Nous sommes le :

    selon le serveur ...
    *** lunedì 20 agosto 2007 15.19.49 CEST *** (IT) ***

  </body>
</html>
```

L'appel du même JSP par une machine distante au sein de l'application Web déployée avec un war donne le même résultat :



JSP utilisant une balise personnalisée - Mozilla Firefox

File Edit View Go Bookmarks Tools Help

Getting Started Latest Headlines

Bienvenue :-)

Et bonjour ! Nous sommes le : selon le serveur ... *** lunedì 20 agosto 2007 15.51.21 CEST *** (IT) ***

Done

This screenshot shows a Mozilla Firefox window displaying a JSP page. The title bar says 'JSP utilisant une balise personnalisée - Mozilla Firefox'. The address bar shows 'http://192.168.1.4:8080/JSPBalises/WelcomeTagDate.jsp'. The main content area displays the rendered HTML with the personalized tag output. The status bar at the bottom says 'Done'.

Comme l'attribut de notre balise n'est pas obligatoire, son absence fera que c'est la version américaine qui sera utilisée :



JSP utilisant une balise personnalisée - Mozilla Firefox

Fichier Édition Affichage Historique Marque-pages Outils ?

http://localhost:8084/JSPBalises/WelcomeTagDate.jsp

Hotmail Personnaliser les liens Windows Media Windows Le Soir en ligne - la_une

Sun SimpleTag (Java 2 Platform Ent. Ed. v... JSP utilisant une balise persona...

Bienvenue :-)

Et bonjour ! Nous sommes le : (selon le serveur) *** Monday, August 20, 2007 4:19:05 PM CEST *** (null) ***

Terminé

This screenshot shows a Mozilla Firefox window displaying a JSP page. The title bar says 'JSP utilisant une balise personnalisée - Mozilla Firefox'. The address bar shows 'http://localhost:8084/JSPBalises/WelcomeTagDate.jsp'. The main content area displays the rendered HTML with the personalized tag output. The status bar at the bottom says 'Terminé'.

12.7 Configuration avec Tomcat comme serveur Web

Avec Tomcat installé sur une machine Windows et si l'on ne désire pas utiliser de war, on peut reproduire la situation existante au sein de l'EDI, soit :

- ◆ le fichier JSPUseTagDate.jsp dans Program Files\Apache Group\Tomcat 4.1\webapps\ROOT
- ◆ le fichier outil.tld dans le répertoire WEB-INF\tlds;
- ◆ le package outil avec BaliseDateDuJour.class dans le sous-répertoire OutilBalises de classes.

12.8 Le fichier web.xml et les librairies de balises

Il est possible de placer dans le fichier web.xml du module web une directive qui permet de créer un alias pour le chemin du fichier des balises personnalisées :

web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
...
<!DOCTYPE web-app
PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
"http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">

<web-app>
<session-config>
<session-timeout>
    5
</session-timeout>
</session-config>
<welcome-file-list>
    ...
</welcome-file-list>

<taglib>
    <taglib-uri>/OutilsVil</taglib-uri>
    <taglib-location>/WEB-INF/tldsoutil.tld</taglib-location>
</taglib>
</web-app>
```

Dans ces conditions, le JSP se contente d'utiliser l'alias dans sa directive `<% @taglib %>` :

```
<% @taglib uri="/OutilsVil" prefix="outil" %>
```

13. Les balises personnalisées avec corps

13.1 Le JSP attendu

Nous allons à présent développer une balise qui possède cette fois un corps, lequel devra être évalué. Ce corps contiendra une commande SQL de type SELECT d'accès à une table (disons "stocks") d'une base de données (connue par son nom de source de données ODBC "marie"). Le JSP qui l'utilisera ressemblera à ceci :

JSPUseTagSelect.jsp

```
<%@page contentType="text/html"%>
<%@taglib uri="/WEB-INF/outilSql.tld" prefix="outilSql" %><head><title>JSP Page -
utilisation de select</title></head>
<body>

<%-- <jsp:useBean id="beanInstanceName" scope="session" class="package.class" /> --%>
<%-- <jsp:getProperty name="beanInstanceName" property="propertyName" /> --%>

<B>Lecture dans la table stocks<b>

<outilSql:select>
    select * from stocks;
</outilSql:select>

<p><B><u>liste obtenue : </u></b><p><p>
<b>codeSto</b> --> <I>quantite</i><p>

<% while (rs.next())
{%
<%= rs.getString("codeSto") %>
et
<%=rs.getInt("quantite") %><p>
<% }%>

<B><I>Fin des opérations ... </I></B>

</body>
</html>
```

Pour écrire ce JSP, il a été supposé que :

- ◆ le fichier tld se nomme **outilSql.tld** et que le préfixe de notre balise est **outilSql**;
- ◆ tout ce qui concerne la balise se trouve dans WEB-INF (nous aurions pu placer tout cela dans un sous-répertoire **tlds**);
- ◆ la balise personnalisée se nomme **select**;
- ◆ la balise initialise une variable de scriptlet nommée **rs**, qui représente manifestement le ResultSet correspondant à la commande SQL passée.

Voyons à présent comment obtenir un tel résultat.

13.2 La classe balise avec corps

Nous allons tout d'abord écrire la classe, appelons-la **SelectTag**, dont le rôle sera d'encapsuler la logique de notre balise. Cette classe doit cette fois implémenter l'interface **BodyTag**, défini dans le package javax.servlet.jsp.tagext, et dérivé de l'interface **Tag** déjà connu; bien sûr, il ajoute à celui-ci les deux méthodes de traitement du corps de la balise, soit :

```
public void doInitBody() throws JspException  
public int doAfterBody() throws JspException
```

dont le rôle semble assez clair. A nouveau, il existe une classe **BodyTagSupport**, dérivée de **TagSupport**, qui fournit une implémentation par défaut de cet interface.

Bien sûr, la méthode **doStartTag()** devra cette fois, en l'absence de problème, retourner la valeur

```
public static final int EVAL_BODY_TAG [deprecated]
```

ou plutôt

```
public static final int EVAL_BODY_BUFFERED
```

définie dans **BodyTag** afin de forcer l'évaluation du corps de la balise tandis que la méthode **doEndTag** retournera, en l'absence d'erreur, la valeur habituelle **EVAL_PAGE**.

Notre classe **SelectTag** héritera donc de **BodyTag Support**. Il nous faudra donc redéfinir :

- ◆ la méthode **doStartTag()** qui se chargera de la connexion à la base de données par un pont JDBC;
- ◆ la méthode **doAfterBody()** qui effectuera la requête de sélection et qui fournira le **ResultSet** rs dont il est fait usage dans le JSP.

On peut cependant se demander comment ce **ResultSet** va être rempli des tuples sur lesquels le JSP lance des regards concupiscents et sera ensuite effectivement transmis au dit JSP. En fait, il existe une classe **BodyContent**, du package javax.servlet.jsp.tagext et dérivée de **JspWriter**, qui permet d'"écrire" dans le JSP mais aussi d'extraire le contenu. Un tel objet, que l'on peut obtenir au moyen de la méthode de **BodyTagSupport** :

```
public BodyContent getBodyContent()
```

peut acquérir le corps de la balise (donc, ici, la chaîne de caractères "select * from stocks") au moyen de sa méthode :

```
public abstract String getString()
```

Il devient ainsi possible de faire exécuter la requête du corps au moyen d'un **executeQuery()**. Il reste alors à appeler la méthode du **pageContext** :

```
public abstract void setAttribute(java.lang.String name, java.lang.Object attribute)
```

qui rendra notre curseur rs accessible au sein de la page sous forme d'une *variable de script* – nous allons y revenir. Cela donne donc :

SelectTag.java

```
import javax.servlet.jsp.tagext.TagSupport;
import javax.servlet.jsp.tagext.BodyTagSupport;
...
import java.sql.*;

public class SelectTag extends BodyTagSupport
{
    Statement instruc;
    boolean arret;

    public SelectTag()
    {
        super();
        instruc = null; arret = false;
    }

    public int doStartTag() throws JspException
    {
        try
        {
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            Connection con = DriverManager.getConnection("jdbc:odbc:marie");
            instruc = con.createStatement();
        }
        catch (ClassNotFoundException e)
        {
            arret = true;
            System.out.println("Classe driver non trouvée !!! : " + e.getMessage());
            return SKIP_BODY
        }
        catch (SQLException e)
        {
            arret = true;
            System.out.println("Erreur SQL !!! : " + e.getMessage() + " - " + e.getSQLState());
            return SKIP_BODY
        }
        return EVAL_BODY_TAG;
    }

    public int doEndTag() throws JspException
    {
        if (!arret) { return EVAL_PAGE; }
        else { return SKIP_PAGE; }
    }
}
```

```

public int doAfterBody() throws JspException, JspException
{
    BodyContent bodyContent = getBodyContent();
    String requete = bodyContent.getString();      // récupération de la requête SQL
    bodyContent.clearBody();
    ResultSet rs = null;
    try
    {
        rs = instruc.executeQuery(requete);
        pageContext.setAttribute("rs",rs);
    }
    catch (SQLException e)
    {
        arret = true;
        System.out.println("Erreur SQL !!! : " + e.getMessage() + " - " + e.getSQLState());
    }
    return SKIP_BODY;
}
}

```

On remarquera l'appel de la méthode

public void clearBody()

afin d'éviter une réévaluation accidentelle lors de la réutilisation du JSP.

Mais comment la variable de script rs, matérialisant le ResultSet, est-elle effectivement reconnue comme étant un tel ResultSet ? Parce qu'un objet fournit cette information ... ou simplement grâce à la TLD.

13.3 Les classes associées aux variables de script

Une classe dérivée de **TagExtraInfo** (on parle encore d'une variable **TEI**), du package javax.servlet.jsp.tagext, a pour rôle de fournir un objet qui renseigne sur la nature de la variable de script associée à une balise donnée. En définitive, elle est fort simple puisqu'elle doit essentiellement redéfinir la méthode

public VariableInfo[] getVariableInfo(TagData data)

La valeur retournée est donc un tableau d'objets (puisque il pourrait y avoir plusieurs variables de script pour le même tag) qui sont des instances de la classe **VariableInfo**, du même package et dont le constructeur est

public VariableInfo(String varName, String className, boolean declare, int scope)

Les deux premiers paramètres sont logiques : il s'agit du nom de la variable de script, soit ici "rs", et de son type, soit ici "ResultSet" (avec précision du package). Le troisième paramètre est, en principe, toujours à true pour signifier que l'on crée ici une nouvelle variable. Le dernier paramètre est la portée de cette variable; la valeur ce paramètre peut être :

public static final int **NESTED**

la variable est seulement visible entre les tags start et end;

public static final int **AT_BEGIN**

la variable est visible après le tag start

public static final int **AT_END**

la variable est visible après le tag end

Dans notre cas, nous retournerons un tableau avec un seul élément qui sera

```
new VariableInfo("rs", "java.sql.ResultSet", true, VariableInfo.AT_END)
```

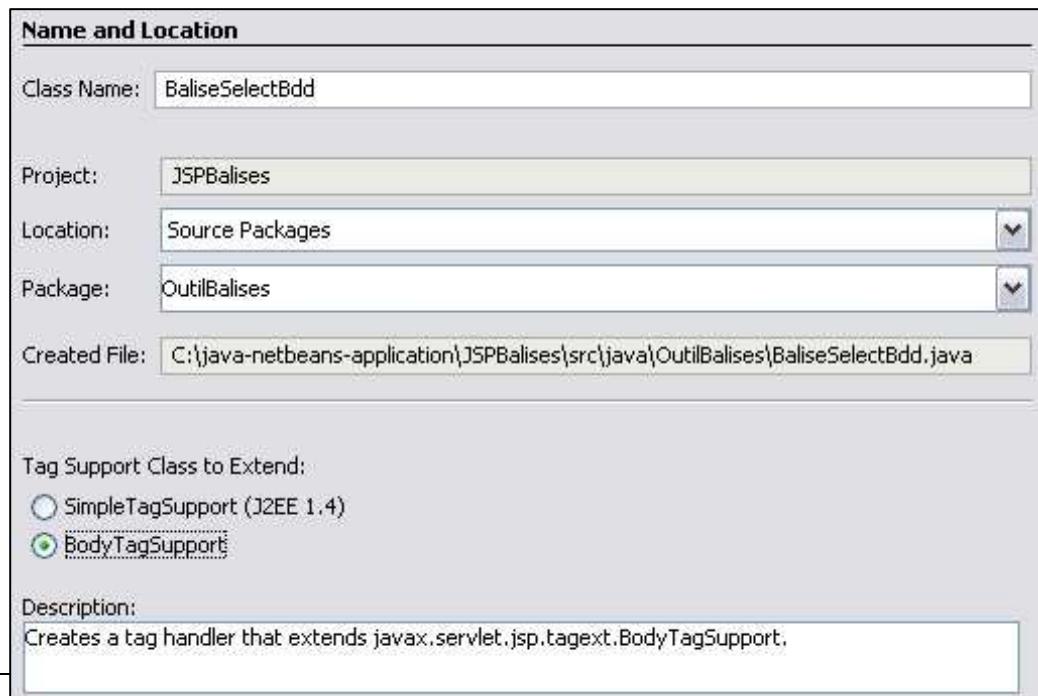
En définitive, la classe associée à notre variable de script rs sera :

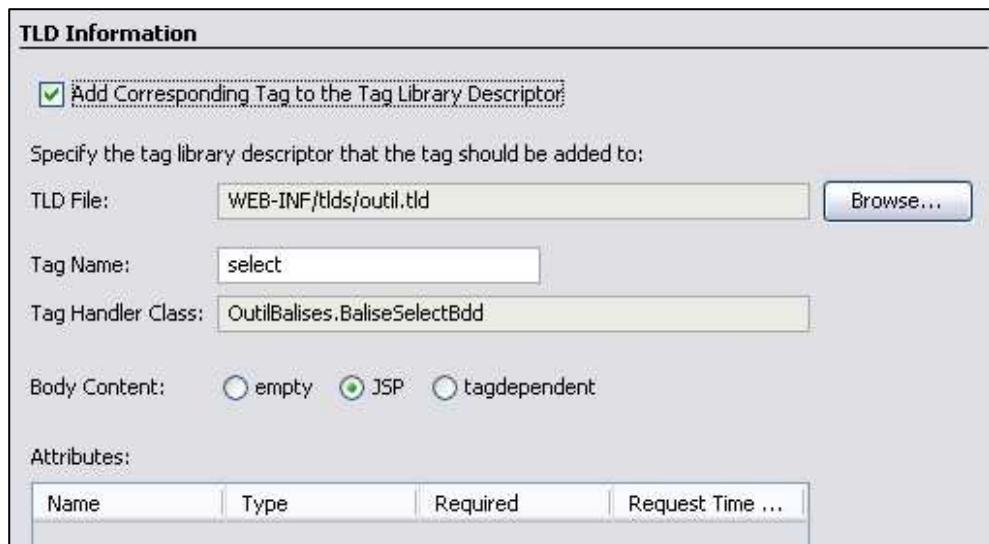
SelectTagVarScript.java

```
import javax.servlet.jsp.tagext.*;  
  
public class SelectTagVarScript extends javax.servlet.jsp.tagext.TagExtraInfo  
{  
    public SelectTagVarScript() {}  
  
    public VariableInfo[] getVariableInfo(TagData data)  
    {  
        return new VariableInfo[]  
        {  
            new VariableInfo("rs", "java.sql.ResultSet", true, VariableInfo.AT_END)  
        };  
    }  
}
```

13.4 La création du tag handler

Dans l'environnement NetBeans, nous pouvons générer la classe associée à la balise :





On obtient cette fois une classe générée plus compliquée :

BaliseSelectBdd.java (résultat de la génération)

```
/*
 * BaliseSelectBdd.java
 */
package OutilBalises;

import java.io.IOException;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.JspWriter;
import javax.servlet.jsp.tagext.BodyContent;
import javax.servlet.jsp.tagext.BodyTagSupport;

/**
 * Generated tag handler class.
 * @author Vilvens
 */

public class BaliseSelectBdd extends BodyTagSupport
{
    /** Creates new instance of tag handler */
    public BaliseSelectBdd() { super(); }

    /////////////////////////////////
    /**
     * User methods.
     */
    /**
     * Modify these methods to customize your tag handler.
     */
    ///////////////////////////////

    /**
     * Method called from doStartTag().
     */
}
```

```
* Fill in this method to perform other operations from doStartTag().  
*  
*/  
private void otherDoStartTagOperations()  
{  
    //  
    // TODO: code that performs other operations in doStartTag  
    // should be placed here.  
    // It will be called after initializing variables,  
    // finding the parent, setting IDREFs, etc, and  
    // before calling theBodyShouldBeEvaluated().  
    //  
    // For example, to print something out to the JSP, use the following:  
    //  
    // try {  
    //     JspWriter out = pageContext.getOut();  
    //     out.println("something");  
    // } catch (IOException ex) {  
    //     // do something  
    // }  
    //  
}  
  
/**  
 * Method called from doEndTag()  
 * Fill in this method to perform other operations from doEndTag().  
 *  
 */  
private void otherDoEndTagOperations()  
{  
    //  
    // TODO: code that performs other operations in doEndTag  
    // should be placed here.  
    // It will be called after initializing variables,  
    // finding the parent, setting IDREFs, etc, and  
    // before calling shouldEvaluateRestOfPageAfterEndTag().  
    //  
}  
  
/**  
 * Fill in this method to process the body content of the tag.  
 * You only need to do this if the tag's BodyContent property  
 * is set to "JSP" or "tagdependent."  
 * If the tag's bodyContent is set to "empty," then this method  
 * will not be called.  
 */  
private void writeTagBodyContent(JspWriter out, BodyContent bodyContent)  
    throws IOException  
{  
    // TODO: insert code to write html before writing the body content.
```

```

// e.g.:
//
// out.println("<strong>" + attribute_1 + "</strong>");
// out.println(" <blockquote>");

//
// write the body content (after processing by the JSP engine) on the output Writer
//
bodyContent.writeOut(out);

//
// Or else get the body content as a string and process it, e.g.:
// String bodyStr = bodyContent.getString();
// String result = yourProcessingMethod(bodyStr);
// out.println(result);
//

// TODO: insert code to write html after writing the body content.
// e.g.:
//
// out.println(" </blockquote>");

// clear the body content for the next time through.
bodyContent.clearBody();
}

///////////////////////////////
///                      ///
/// Tag Handler interface methods.          ///
///                      ///
/// Do not modify these methods; instead, modify the   ///
/// methods that they call.                     ///
///                      ///
///////////////////////////////

/**
 * This method is called when the JSP engine encounters the start tag,
 * after the attributes are processed.
 * Scripting variables (if any) have their values set here.
 * @return EVAL_BODY_BUFFERED if the JSP engine should evaluate the tag body,
otherwise return SKIP_BODY.
 * This method is automatically generated. Do not modify this method.
 * Instead, modify the methods that this method calls.
 */

public int doStartTag() throws JspException, JspException
{
    otherDoStartTagOperations();
}

```

```

if (theBodyShouldBeEvaluated())
{
    return EVAL_BODY_BUFFERED;
}
else
{
    return SKIP_BODY;
}

/***
 * This method is called after the JSP engine finished processing the tag.
 * @return EVAL_PAGE if the JSP engine should continue evaluating the JSP page,
otherwise return SKIP_PAGE.
 * This method is automatically generated. Do not modify this method.
 * Instead, modify the methods that this method calls.
*/
public int doEndTag() throws JspException, JspException
{
    otherDoEndTagOperations();

    if (shouldEvaluateRestOfPageAfterEndTag())
    {
        return EVAL_PAGE;
    }
    else
    {
        return SKIP_PAGE;
    }
}

/***
 * This method is called after the JSP engine processes the body content of the tag.
 * @return EVAL_BODY AGAIN if the JSP engine should evaluate the tag body again,
otherwise return SKIP_BODY.
 * This method is automatically generated. Do not modify this method.
 * Instead, modify the methods that this method calls.
*/
public int doAfterBody() throws JspException
{
    try
    {
        //
        // This code is generated for tags whose bodyContent is "JSP"
        //
        BodyContent bodyContent = getBodyContent();
        JspWriter out = bodyContent.getEnclosingWriter();

        writeTagBodyContent(out, bodyContent);
    }
}

```

```
catch (Exception ex)
{
    handleBodyContentException(ex);
}

if (theBodyShouldBeEvaluatedAgain())
{
    return EVAL_BODY AGAIN;
}
else
{
    return SKIP_BODY;
}

/**
 * Handles exception from processing the body content.
 */
private void handleBodyContentException(Exception ex) throws JspException
{
    // Since the doAfterBody method is guarded, place exception handing code here.
    throw new JspException("error in NewTag: " + ex);
}

/**
 * Fill in this method to determine if the rest of the JSP page
 * should be generated after this tag is finished.
 * Called from doEndTag().
 */
private boolean shouldEvaluateRestOfPageAfterEndTag()
{
    //
    // TODO: code that determines whether the rest of the page
    // should be evaluated after the tag is processed
    // should be placed here.
    // Called from the doEndTag() method.
    //
    return true;
}

/**
 * Fill in this method to determine if the tag body should be evaluated
 * again after evaluating the body.
 * Use this method to create an iterating tag.
 * Called from doAfterBody().
 */
private boolean theBodyShouldBeEvaluatedAgain()
{
    //
    // TODO: code that determines whether the tag body should be
```

```

// evaluated again after processing the tag
// should be placed here.
// You can use this method to create iterating tags.
// Called from the doAfterBody() method.
//
return false;
}

/**
 * Fill in this method to determine if the tag body should be evaluated.
 * Called from doStartTag().
 */
private boolean theBodyShouldBeEvaluated()
{
//
// TODO: code that determines whether the body should be
// evaluated should be placed here.
// Called from the doStartTag() method.
//
return true;
}

```

Nous pouvons à présent tailler ce template selon nos souhaits pour obtenir :

BaliseSelectBdd.java (après adaptation)

```

package OutilBalises;

import javax.servlet.jsp.tagext.TagSupport;
import javax.servlet.jsp.tagext.BodyTagSupport;
import javax.servlet.jsp.tagext.BodyContent;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.JspWriter;
import javax.servlet.jsp.PageContext;
import javax.servlet.ServletRequest;
import java.io.PrintWriter;
import java.io.IOException;
import java.sql.*;

public class BaliseSelectBdd extends BodyTagSupport
{
    Statement instruc;
    boolean arret;
    public BaliseSelectBdd()
    {
        super();
        instruc = null;
        arret = false;
    }
}
```

```
public void otherDoStartTagOperations()  
{  
    try  
    {  
        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");  
        Connection con = DriverManager.getConnection("jdbc:odbc:marie");  
        instruc = con.createStatement();  
    }  
    catch (ClassNotFoundException e)  
    {  
        arret = true;  
        System.out.println("Classe driver non trouvée !!! : " + e.getMessage());  
    }  
    catch (SQLException e)  
    {  
        arret = true;  
        System.out.println("Erreur SQL !!! : " + e.getMessage() + " - " + e.getSQLState());  
    }  
}  
  
public boolean theBodyShouldBeEvaluated()  
{  
    if (arret) return false;  
    else return true;  
}  
  
public void otherDoEndTagOperations() { }  
  
public boolean shouldEvaluateRestOfPageAfterEndTag()  
{  
    if (arret) return false;  
    else return true;  
}  
  
public boolean theBodyShouldBeEvaluatedAgain()  
{  
    return false;  
}  
  
public int doStartTag() throws JspException, JspException  
{  
    otherDoStartTagOperations();  
  
    if (theBodyShouldBeEvaluated()) { return EVAL_BODY_TAG; }  
    else { return SKIP_BODY; }  
}  
  
public int doEndTag() throws JspException, JspException  
{  
    otherDoEndTagOperations();
```

```

if (shouldEvaluateRestOfPageAfterEndTag()) { return EVAL_PAGE; }
else { return SKIP_PAGE; }
}

public int doAfterBody() throws JspException, JspException
{
    try
    {
        JspWriter out = getPreviousOut();
        BodyContent bodyContent = getBodyContent();
        writeTagBodyContent(out, bodyContent);
    }
    catch (Exception ex) { throw new JspException("error in SelectTag: " + ex); }

    if (theBodyShouldBeEvaluatedAgain()) { return EVAL_BODY_TAG; }
    else { return SKIP_BODY; }
}

public void writeTagBodyContent(JspWriter out, BodyContent bodyContent)
throws IOException
{
    String requete = bodyContent.getString();
    ResultSet rs = null;
    try
    {
        rs = instruc.executeQuery(requete);
        pageContext.setAttribute("rs",rs);
    }
    catch (SQLException e)
    {
        arret = true;
        System.out.println("Erreur SQL !!! : " + e.getMessage() + " - " + e.getSQLState());
    }

    bodyContent.writeOut(out);
    bodyContent.clearBody();
}
}

```

13.5 Le TLD pour une balise personnalisée avec corps

Pour notre exemple, nous avons utilisé le même fichier tld que pour la balise sans coprs. Il ressemblera à ceci :

outil.tld (avec la description des deux balises)

```

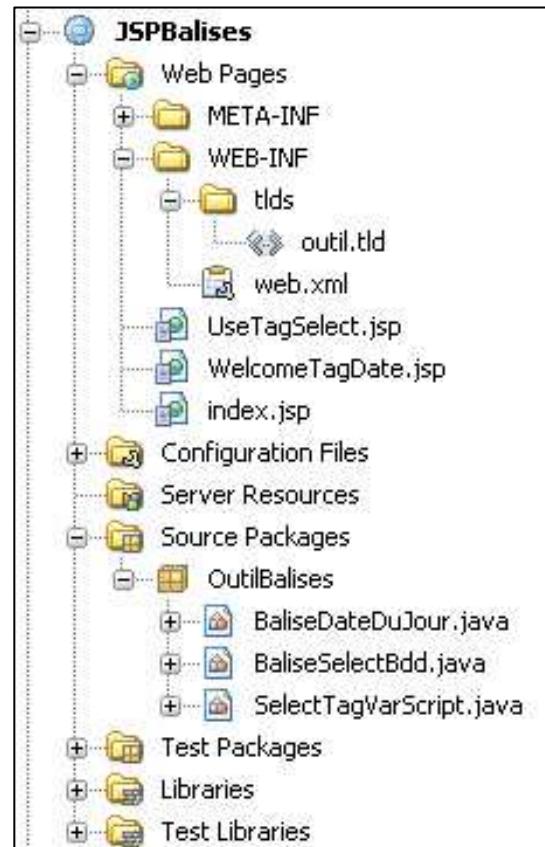
<?xml version="1.0" encoding="UTF-8"?>
<taglib version="2.0" xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee web-jsptaglibrary_2_0.xsd">
<tlib-version>1.0</tlib-version>

```

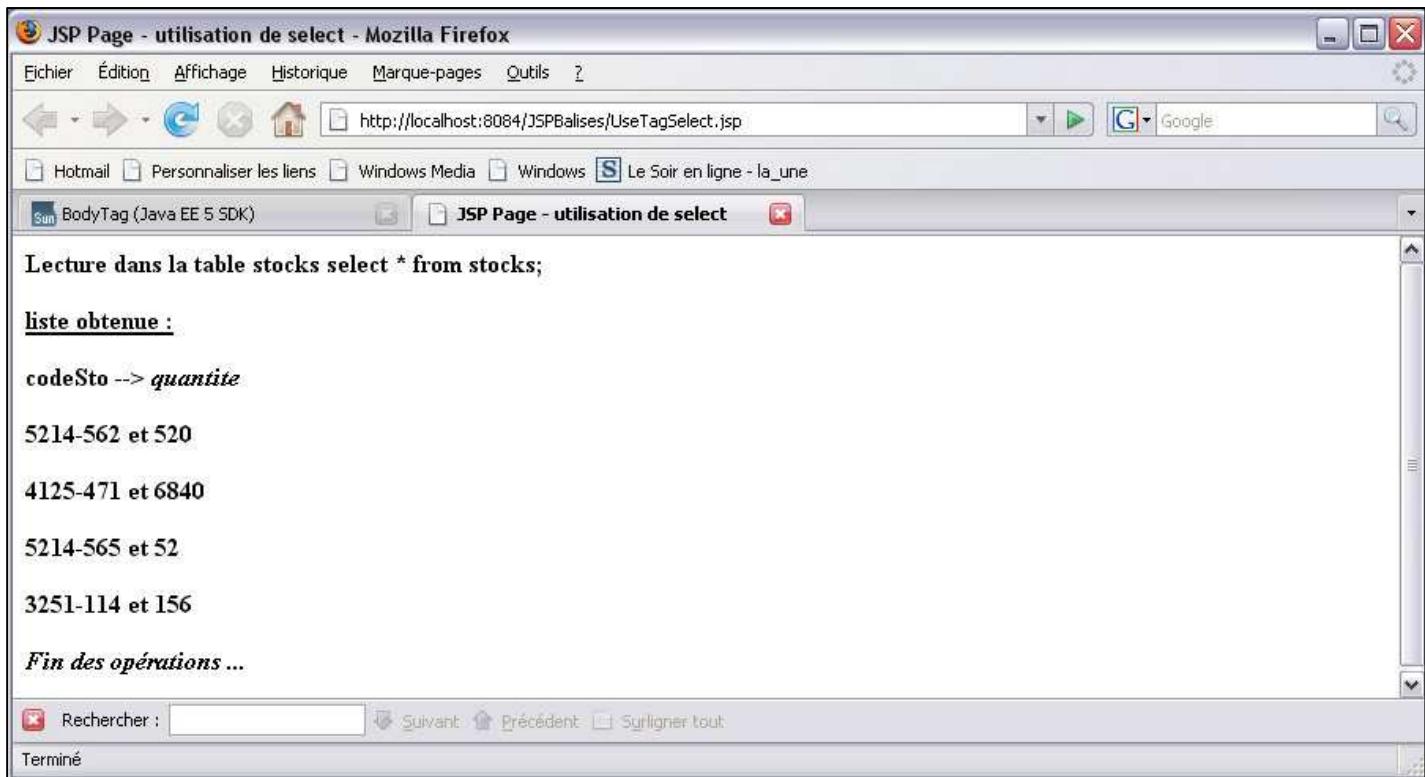
```
<short-name>outil</short-name>
<uri>/WEB-INF/tlds/outil</uri>
<tag>
  <name>maintenant</name>
  <tag-class>OutilBalises.BaliseDateDuJour</tag-class>
  <body-content>empty</body-content>
  <attribute>
    <name>langue</name>
    <rtextrvalue>true</rtextrvalue>
    <type>java.lang.String</type>
  </attribute>
</tag>
<tag>
  <name>select</name>
  <tag-class>OutilBalises.BaliseSelectBdd</tag-class>
  <body-content>JSP</body-content>
  <variable>
    <name-given>rs</name-given>
    <variable-class>java.sql.ResultSet</variable-class>
    <declare>true</declare>
    <scope>AT_END</scope>
  </variable>
</tag>
</taglib>
```

13.6 L'exécution du JSP utilisant la balise avec corps

Sous NetBeans, le projet ressemble en définitive à ceci :



Une exécution depuis NetBeans en local pourrait donner :



tandis que la même application Web depuis une machine distante (attention : la source de données doit être système) :



Remarque

Notre première balise sans corps (classe BaliseDateDuJour) a été écrite à partir d'une dérivation de la classe SimpleTagSupport. Mais nous aurions pu aussi l'écrire, à l'image de la balise avec corps (classe BaliseSelectBdd) à partir de la classe TagSupport, très proche de la classe BodyTagSupport. Cela aurait donné :

BaliseDateDuJour.java (code avec TagSupport)

```
package outil;

import javax.servlet.jsp.tagext.Tag;
...
import java.util.*;
import java.text.*;

public class BaliseDateDuJour extends TagSupport
{
    private String langue = "FR";
    private String chDate;

    public BaliseDateDuJour() { super(); }

    public void otherDoStartTagOperations()
    {
        try
        {
            pageContext.getOut().println("(selon le serveur)"); // pour dire quelque chose ...
        }
        catch (IOException e) {}
    }

    public boolean theBodyShouldBeEvaluated() { return false; }

    public void otherDoEndTagOperations()
    {
        Date maintenant = new Date();
        if (langue == "UK")
            chDate = DateFormat.getDateTimeInstance(DateFormat.FULL,DateFormat.FULL,
                Locale.UK).format(maintenant);
        else
            if (langue == "IT")
                chDate = DateFormat.getDateTimeInstance(DateFormat.FULL,DateFormat.FULL,
                    Locale.ITALY).format(maintenant);
            else
                if (langue == "FR ")
                    chDate = DateFormat.getDateTimeInstance(DateFormat.FULL,DateFormat.FULL,
                        Locale.FRANCE).format(maintenant);
                else

```

```
chDate = DateFormat.getDateTimeInstance(DateFormat.FULL,DateFormat.FULL,
                                         Locale.US).format(maintenant);
try
{
    pageContext.getOut().println("*** " + chDate + " *** (" + getLangue() + ") ***");
}
catch (IOException e) {}

public boolean shouldEvaluateRestOfPageAfterEndTag() { return true; }

public int doStartTag() throws JspException, JspException
{
{
    otherDoStartTagOperations();

    if (theBodyShouldBeEvaluated()) { return EVAL_BODY_INCLUDE; }
    else { return SKIP_BODY; }
}

public int doEndTag() throws JspException, JspException
{
    otherDoEndTagOperations();

    if (shouldEvaluateRestOfPageAfterEndTag()) { return EVAL_PAGE; }
    else { return SKIP_PAGE; }
}

public String getLangue() { return langue; }

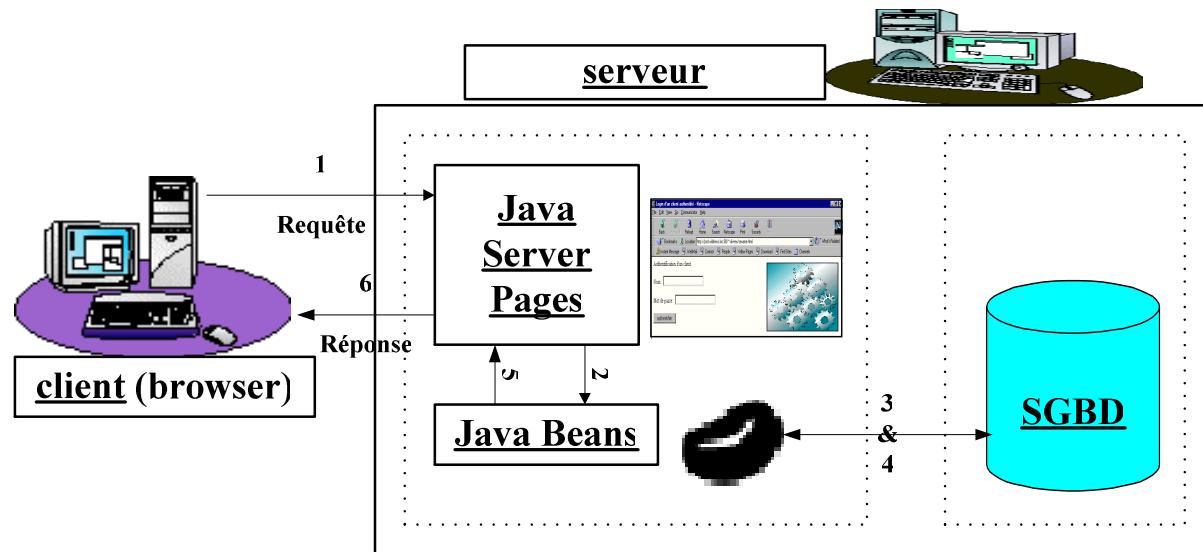
public void setLangue(String value) { langue = value; }
}
```

14. Les modèles d'architecture à base de JSP

Au point où nous en sommes, il est sans doute utile de résumer les possibilités de construction d'une application Web basée sur les JSP. En fait, on distingue deux architectures possibles.

14.1 Le modèle 1 pour applications simples

Le JSP y sert effectivement de frontal pour le client : il reçoit la requête de celui-ci et lui enverra la réponse. L'accès aux données (par appel à un SGBD ou autre chose⁹) s'effectue par un (ou plusieurs) bean(s) au(x)quel(s) le JSP fera appel. Comme nous l'avons déjà signalé, on constate qu'il y a séparation entre la présentation et le contenu.



A priori, ceci semble suffire. Cependant, ce modèle montre ses limites pour des applications plus complexes. En effet, dans ce cas,

- ◆ l'ensemble des JSPs contiendra alors un nombre imposant de scriptlets, compliquant le travail des designers non-programmeurs;
- ◆ il y a confusion entre la présentation et logique de l'application (autre que ce qui concerne l'accès aux données, puisque ceci est pris en compte par les Java Beans).

Cette dernière remarque conduit au second modèle ...

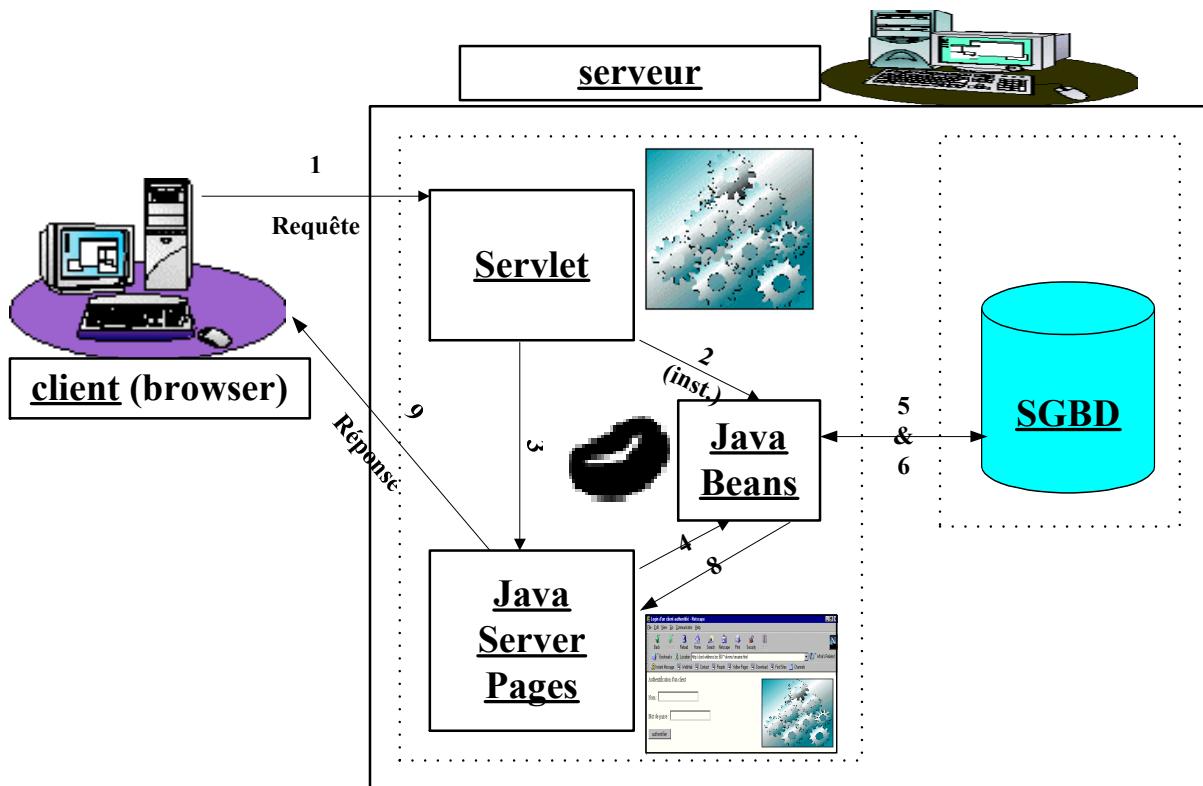
14.2 Le modèle 2 pour applications complexes

Ce modèle 2 va donc séparer l'aspect visuel (encodage de la requête et présentation de la réponse) du traitement proprement dit effectué sur les données fournies par les beans. C'est le moment de se souvenir que si les JSP sont dédiés à la présentation, les servlets sont plutôt vouées au traitement : on va donc imaginer une servlet dont le rôle sera de

- ◆ éventuellement servir de frontal pour le client (mais un JSP peut aussi débuter le dialogue);
 - ◆ effectuer les traitements et instancier les beans d'accès aux données nécessaires;
 - ◆ surtout aiguiller les résultats, intermédiaires ou finaux, vers le JSP concerné selon les actions du client.

⁹ c'est-à-dire un EJB – voir plus loin

Le JSP reprend alors la même fonctionnalité que dans le modèle 1 : extraction des données fournies par un Java Bean et présentation de la réponse au client.



La servlet joue donc le rôle de **contrôleur**. Le JSP n'a plus alors comme rôle que de fournir au client un **vue** des données, vue extraite d'un **modèle** de données par le bean. On parle encore pour cette raison de modèle **MVC (Model-View-Controller)**¹⁰.

Si, de plus, les JSP font usage des balises personnalisées, les infographistes et les développeurs seront tous deux comblés :

- les infographistes n'auront qu'à se préoccuper de l'aspect design de présentation de l'information (pages HTML statiques, JSP avec "boîtes noires" en forme de tags pour les pages dynamiques, feuilles de style CSS pour une évolutivité aisée du site);
- les programmeurs ne devront plus guère mettre au point du code Java au sein des JSP : ils développeront et débogueront des servlets et des beans, donc des classes Java classiques.

Ne toucherions-nous pas ici au nirvana des sociétés de développement de solutions WEB ;-) ? A dire vrai, il manque encore les Enterprise Java Beans – nous y ferons allusion en fin de volume.

¹⁰ Ce modèle MVC fait à présent partie des "Design patterns", c'est-à-dire des modèles de programmation reconnus comme des standards quels que soient le langage et la plate-forme utilisés.

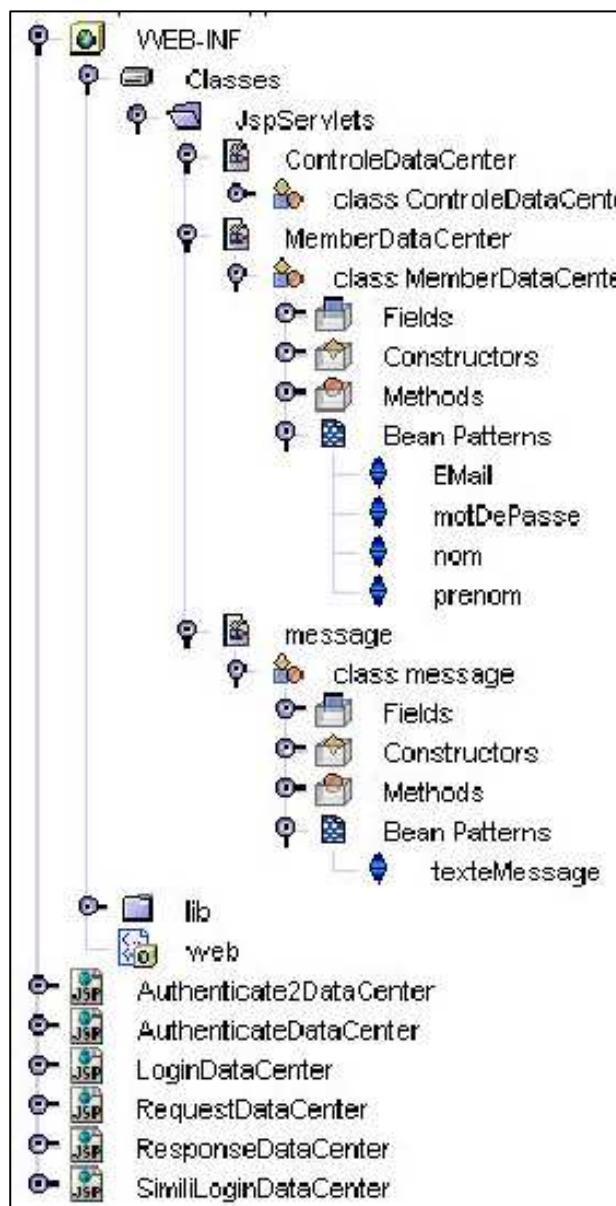
15. L'implémentation pratique d'un modèle MVC

15.1 L'application Web DataCenter

Afin d'illustrer le modèle MVC, considérons un Data Center qui fournit des informations bibliographiques sur des auteurs scientifiques. A priori, ces informations sont accessibles à tout le monde, pour peu que le client fasse l'effort de s'enregistrer (on peut imaginer plus compliqué, mais bon, ce sera pour une autre fois ;-)).

Notre application Web va comporter les éléments suivants :

- ◆ une **servlet de contrôle** ControleDataCenter.java appartenant au package JspServlets;
- ◆ des **Java Server Pages correspondant à chaque étape** d'utilisation du client : LoginDataCenter.jsp, RequestDataCenter.jsp, etc;
- ◆ un **Java Bean** MemberDataCenter.java qui représentera les coordonnées du client en cours;
- ◆ un autre **Java Bean** message.java qui représentera la pensée du jour pour tous les clients.



Voyons plus précisément de quels éléments programmatiques nous avons besoin.

La servlet va donc constituer le point de passage obligé entre deux JSPs successifs : autrement dit, le client ne passera jamais directement d'un JSP à un autre. De cette manière, la servlet pourra jouer à plein son rôle de contrôleur. Pour passer d'une servlet à une autre (on peut parler ainsi, puisqu'un JSP se traduit finalement en servlet), nous connaissons déjà la méthode de `HttpServletResponse` :

```
public abstract void sendRedirect(String location) throws IOException
```

mais l'URL utilisée doit être absolue, du moins dans les spécification des servlets antérieures à 2.2. Nous utiliserons ici une autre méthode, qui nous sera accessible grâce au `ServletContext`, que l'on peut toujours obtenir au moyen de la méthode `getServletContext()`. En effet, ce `ServletContext` comporte une méthode :

```
public RequestDispatcher getRequestDispatcher(java.lang.String path)
```

L'objet `RequestDispatcher` que l'on obtient ainsi représente en fait non pas la ressource (servlet, JSP, page HTML) désignée par le paramètre mais son "enveloppe" ("wrapper" en anglais), c'est-à-dire qu'elle renferme la référence à cette ressource, avec le moyen de charger cette ressource dans le browser client au moyen de sa méthode :

```
public void forward(ServletRequest request, ServletResponse response)  
throws ServletException, java.io.IOException
```

Le chemin précisé dans `getRequestDispatcher()` est un chemin relatif au contexte (donc, pour Tomcat, ROOT de webapps); il doit toujours débuter par le caractère "/". On peut voir que la méthode `forward()` reprend les objets requête et réponse, permettant ainsi un passage facile d'une servlet à une autre ... Il existe même une action de ce type pour les JSP :

```
<jsp:forward page=nom_de_la_page.jsp />
```

15.2 Les données partagées entre JSPs et leur portée

Si il est vrai que le passage d'un JSP à un autre (ou, dans notre cas, à une servlet) semble aisément, il faut cependant bien voir que seuls les paramètres envoyés explicitement par la servlet émettrice seront transmis à la servlet réceptrice : pas question d'imaginer une chaîne de servlets avec des paramètres qui se propageraient depuis le début jusqu'à la fin (dans un sens, cela vaut d'ailleurs mieux pour éviter une espèce d'inflation des paramètres).

Se pose donc ici le problème de partager des données entre les JSP et la servlet contrôleur de notre application de type MVC. Le moyen nous est en fait connu : un Java bean représentera cette information partagée. Celui-ci peut être mémorisé de différentes manières, ce qui conduit à différentes portées. Par **portée**, on entend la durée pendant laquelle l'objet est disponible et s'il est disponible à un ou à tous les utilisateurs. Les possibilités sont **page**, **requête**, **session** et **application**. En pratique, un Java bean préalablement instancié sera partagé selon une portée :

a) **requête** : si il est enregistré et géré au sein de l'objet **HttpServletRequest** au moyen des méthodes de cet objet requête

```
public void setAttribute(java.lang.String name, java.lang.Object object)  
public java.lang.Object getAttribute(java.lang.String name)  
public void removeAttribute(java.lang.String name)
```

b) session : si il est enregistré et géré au sein d'un objet **HttpServletSession**. Nous avons l'habitude des méthodes

```
public abstract void putValue(String name, Object value)
public abstract Object getValue(String name)
public void removeValue(java.lang.String name)
```

- mais, pour rappel, celles-ci sont déclarées obsolètes dans les versions récentes du JDK et sont remplacées par

```
public void setAttribute(java.lang.String name, java.lang.Object object)
public java.lang.Object getAttribute(java.lang.String name)
public void removeAttribute(java.lang.String name)
```

Oui, c'est bien cela : ce sont les mêmes prototypes que pour l'objet requête. D'ailleurs ...

c) application : si il est enregistré pour l'ensemble de l'application Web, c'est-à-dire accessible par des servlets qui peuvent avoir été appelées par des utilisateurs différents. Pour ce faire, notre bean sera enregistré dans l'objet **ServletContext** qui possède bien sûr les méthodes :

```
public void setAttribute(java.lang.String name, java.lang.Object object)
public java.lang.Object getAttribute(java.lang.String name)
public void removeAttribute(java.lang.String name)
```

A remarquer que l'on peut obtenir les noms des attributs disponibles grâce à :

```
public java.util.Enumeration getAttributeNames()
```

15.3 La logique de la servlet contrôleur

L'idée générale est que chaque JSP invoque le servlet de contrôle en lui passant un paramètre nommé "**action**", qui peut être un élément visible du formulaire se trouvant dans le JSP ou plutôt un champ caché ("hidden") de ce formulaire. Selon la valeur du paramètre action récupéré, la servlet entame l'une ou l'autre action et transfère le client sur un autre JSP. Schématiquement :

JSP	Paramètre action envoyé <i>(visible ou caché)</i>	Réaction de la servlet
LoginDataCenter	Authentification/ Requête/ Terminer (<i>visibles</i>)	Authentification : création d'un bean MemberDataCenter Requête : renvoi sur LoginDataCenter ou RequestDataCenter Terminer : session supprimée
AuthenticateDataCenter	AuthenticationInProgress (<i>caché</i>)	Mise à jour éventuelle du mot de passe du client
Authenticate2DataCenter	AuthenticateRecorded (<i>caché</i>)	Configuration définitive du bean instance de MemberDataCenter
RequestDataCenter	RequestHandling (<i>caché</i>)	Vérification de l'authentification et renvoi éventuel au login – recherche des références demandées [ici, prrt ...]

ResponseDataCenter	NewRequest (caché)	Selon le paramètre réponse, retour à une nouvelle requête ou aiguillage vers la page finale
SimiliLoginDataCenter	Requête/Terminer (visibles)	voir LoginDataCenter

15.4 Les codes des JSP

Les divers Java Server Pages utilisés sont, sans surprise, les suivants :

LoginDataCenter.jsp

```
<%@page language="java" %>
<%@page contentType="text/html; charset=ISO-8859-1"%>
<%@page info="(c) Claude Vilvens - 8/2004" %>
<%@page import="java.util.*" %>
<%@page import="java.text.*" %>
<html>
<head><title>Login sur le Data Center des références</title></head>

<body>
<%! Date maintenant = new Date(); %>
<%! String laDate =
DateFormat.getTimeInstance(DateFormat.FULL,DateFormat.FULL,Locale.FRANCE).
format(maintenant); %>

<!-- Page demandée le <%=laDate %> -->
<P>Bienvenue sur LE site des références scientifiques !<p>
Nous sommes le <%=laDate %> !!!<p>

<% String msg= request.getParameter("msg");
if (msg!=null) out.println("<H2>Message du serveur : " + msg + "</H2><p>");
%>

Veuillez entrer votre nom et votre mot de passe ...<p>

<form method="POST" action=" http://claude:8081/servlet/ControleDataCenter">
<P>Authentification préalable, requête (après authentification obligatoire) ou fin ?<p>
<SELECT Name="action">
<OPTION>Authentification
<OPTION>Requête
<OPTION>Terminer
</SELECT></P>

<P>Nom : <input type="text" name="nom" size=20></P>
<P>Mot de passe : <input type="password" name="motDePasse" size=20></P>
<P><input type="submit" value="action"></P>
</form>
</body>
</html>
```

AuthenticateDataCenter.jsp

```
<%@page contentType="text/html"%>
<html>
<head><title>Authentification sur le Data Center (1)</title></head>
<body>

<p>Vous souhaitez donc entrer dans notre grande famille de chercheurs - félicitations !<p>

<% String nom = request.getParameter("nom");%>

Vous êtes donc : <%=nom%>

Confirmez votre mot de passe :
<form method="POST" action=" http://claude:8081/servlet/ControleDataCenter">
<P>Mot de passe : <input type="password" name="motDePasse" size=20></P>
<input type="hidden" name="action" value="AuthentificationInProgress">
<P><input type="submit" value="confirmer"></P>
</form>

</body>
</html>
```

Authenticate2DataCenter.jsp

```
<%@page contentType="text/html"%>
<jsp:useBean id="membre" scope="session" class="JspServlets.MemberDataCenter" />
<html>
<head><title>Authentification sur le Data Center (2)</title></head>
<body>

<% String password = request.getParameter("motDePasse");%>

Vous êtes donc : <jsp:getProperty name="membre" property="nom" />
avec le mot de passe : <%=password%>

<p>Nous souhaitons quelques renseignements complémentaires<p>

<form method="POST" action=" http://claude:8081/servlet/ControleDataCenter">
<P>Prénom : <input type="text" name="prenom" size=20></P>
<P>Votre e-mail : <input type="text" name="eMail" size=30></P>
<input type="hidden" name="action" value="AuthenticateRecorded">
<P><input type="submit" value="confirmer"></P>
</form>

</body>
</html>
```

RequestDataCenter.jsp

```
<%@page contentType="text/html"%>
<jsp:useBean id="membre" scope="session" class="JspServlets.MemberDataCenter" />
<jsp:useBean id="msgGeneral" scope="application" class="JspServlets.message" />
<html>
<head><title>Requête au le Data Center (1)</title></head>
<body>

<H3>Bonjour cher <jsp:getProperty name="membre" property="prenom" />
<jsp:getProperty name="membre" property="nom" /> !
</h3>
<p>Je vous écoute : que cherchez-vous ?</p>

<form method="POST" action=" http://claude:8081/servlet/ControleDataCenter">
<P>Auteur : <input type="text" name="auteur" size=20></P>
<P>Année de publication : <input type="text" name="annee" size=30></P>
<input type="hidden" name="action" value="RequestHandling">
<P><input type="submit" value="action"></P>
</form>
<p><hr>
La pensée du jour : <p><jsp:getProperty name="msgGeneral" property="texteMessage" />

</body>
</html>
```

ResponseDataCenter.jsp

```
<%@page contentType="text/html"%>
<jsp:useBean id="msgGeneral" scope="application" class="JspServlets.message" />
<html>
<head><title>Réponse du Data Center (1)</title></head>
<body>

<H3>Voici les références correspondant à votre demande : </H3><p>0 référence ... :-(<p>

<form method="POST" action=" http://claude:8081/servlet/ControleDataCenter">
<P>Autre requête ?
<SELECT Name="reponse">
<OPTION>Oui
<OPTION>Non
</SELECT></P>
<input type="hidden" name="action" value="NewRequest">
<P><input type="submit" value="action"></P>
</form>

<p><hr>
La pensée du jour : <p><jsp:getProperty name="msgGeneral" property="texteMessage" />

</body>
</html>
```

SimiliLoginDataCenter.jsp

```
<%@page language="java" %>
<%@page contentType="text/html; charset=ISO-8859-1"%>
<%@page info="(c) Claude Vilvens - 8/2004" %>
<%@page import="java.util.*" %>
<%@page import="java.text.*" %>
<jsp:useBean id="msgGeneral" scope="application" class="JspServlets.message" />
<html>
<head><title>Data Center des références</title></head>

<body>

<%! Date maintenant = new Date(); %>
<%! String laDate =
DateFormat.getTimeInstance(DateFormat.FULL,DateFormat.FULL,Locale.FRANCE).
format(maintenant); %>

<!-- Page demandée le <%=laDate %> -->
<P>Vous êtes toujours sur LE site des références scientifiques !<p>
Nous sommes le <%=laDate %> !!!<p>

<form method="POST" action=" http://claude:8081/servlet/ControleDataCenter">
<P>Requête (vous êtes déjà authentifié) ou fin ?<p>
<SELECT Name="action">
<OPTION>Requête
<OPTION>Terminer
</SELECT></P>
<P><input type="submit" value="action"></P>
</form>
<p><hr>
La pensée du jour : <p><jsp:getProperty name="msgGeneral" property="texteMessage" />

</body>
</html>
```

15.5 Les codes des deux Java beans

Il s'agit des deux Beans représentant le membre et le emssage général, beans dont nous avons vu trace dans les JSP ci-dessus. Ici non plus, rien d'extraordinaire ...

MemberDataCenter.java

```
package JspServlets;

/*
 * MemberDataCenter.java
 * Created on 4 août 2004, 10:00
 */
/***
 * @author Vilvens
 */
```

```
public class MemberDataCenter
{
    private String nom;
    private String prenom;
    private String motDePasse;
    private String eMail;

    private MemberDataCenter()
    {
        nom = prenom = motDePasse = eMail = null;
    }

    public MemberDataCenter(String n, String pwd)
    {
        nom = n;
        motDePasse = pwd;
        prenom = eMail = null;
    }

    public boolean equals(MemberDataCenter mdc)
    {
        return ( nom.equals(mdc.nom) && prenom.equals(mdc.prenom) &&
                 motDePasse.equals(mdc.motDePasse) );
    }

    public String getNom() { return nom; }
    public String getPrenom() { return prenom; }
    public String getMotDePasse() { return motDePasse; }
    public String getEMail() { return eMail; }

    public void setNom(String x) { nom = x; }
    public void setPrenom(String x) { prenom = x; }
    public void setMotDePasse(String x) { motDePasse = x; }
    public void setEMail(String x) { eMail = x; }
}
```

message.java

```
package JspServlets;

/*
 * message.java
 * Created on 4 août 2004, 17:17
 */

/**
 * @author Vilvens
 */
```

```
public class message
{
    private String texteMessage;

    public message(String m)
    {
        texteMessage = m;
    }

    public String getTexteMessage() { return texteMessage; }

    public void setTexteMessage (String m) { texteMessage = m; }
}
```

15.6 Le code la servlet

Comme déjà dit, la servlet réagit selon le paramètre action. Elle crée deux types d'informations partageables :

- ◆ une information membre de portée session;
- ◆ une information message général de portée application.

ControleDataCenter.java

```
package JspServlets;

/*
 * ControleDataCenter.java
 * Created on 3 août 2004, 14:37
 */

/**
 * @author Vilvens
 * @version
 */

import javax.servlet.*;
import javax.servlet.http.*;

import java.net.*;

public class ControleDataCenter extends HttpServlet
{
    public void init(ServletConfig config) throws ServletException
    {
        super.init(config);
        ServletContext sc = getServletContext();
        sc.log("-- démarrage de la servlet ControleDataCenter");
    }
}
```

```

public void destroy() { }

protected void processRequest(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, java.io.IOException
{
    java.io.PrintWriter out = response.getWriter();
    ServletContext sc = getServletContext();
    message penseeDuJour = new message("A vaincre sans péril, on triomphe sans gloire");
    sc.setAttribute("msgGeneral", penseeDuJour);

    sc.log-- passage par la servlet ControleDataCenter);
    String action = request.getParameter("action");
    sc.log("-- Valeur du paramètre action : " + action);

    if (action.equals("Authentification"))
    {
        setAuthenticationInProgress(request, true);
        sc.log("Nom récupéré par paramètre = " + request.getParameter("nom") + " -- ok ?");
        MemberDataCenter mdc = new MemberDataCenter(request.getParameter("nom"),
            request.getParameter("motDePasse"));
        HttpSession session = request.getSession(true);
        session.setAttribute("membre", mdc);
        RequestDispatcher rd = sc.getRequestDispatcher("/AuthenticateDataCenter.jsp");
        sc.log("-- Tentative de redirection sur AuthenticateDataCenter.jsp");
        rd.forward(request, response);
    }
    else
    if (action.equals("Terminer"))
    {
        HttpSession session = request.getSession(true);
        session.invalidate();
        Object obj = sc.getAttribute("membre");
        if (obj != null) sc.removeAttribute("membre");
    }

    if (!isAuthenticated(request) && !isAuthenticationInProgress(request) )
    {
        sc.log("-- Validation introuvable dans la session");
        RequestDispatcher rd = sc.getRequestDispatcher("/LoginDataCenter.jsp?msg=" +
            URLEncoder.encode("Veuillez d'abord vous authentifier !"));
        rd.forward(request, response);
    }
    else
    if (action.equals("AuthentificationInProgress"))
    {
        HttpSession session = request.getSession(true);
        MemberDataCenter mdc = (MemberDataCenter) session.getAttribute("membre");
        sc.log("Nom récupéré = " + mdc.getNom());
        sc.log("Nouveau mot de passe éventuel : " + request.getParameter("motDePasse"));
        mdc.setMotDePasse(request.getParameter("password"));
    }
}

```

```

session.setAttribute("membre", mdc);
RequestDispatcher rd = sc.getRequestDispatcher("/Authenticate2DataCenter.jsp");
sc.log("-- Tentative de redirection sur Authenticate2DataCenter.jsp");
rd.forward(request, response);
}
else
if (action.equals("Authenticaterecorded"))
{
    HttpSession session = request.getSession(true);
    sc.log("-- Enregistrement du nouveau membre");
    MemberDataCenter mdc = (MemberDataCenter) session.getAttribute("membre");
    sc.log("Nom récupéré = " + mdc.getNom());
    mdc.setPrenom(request.getParameter("prenom"));
    mdc.setEMail(request.getParameter("eMail"));
    session.setAttribute("membre", mdc);
    setAuthenticated(request, true);
    RequestDispatcher rd = sc.getRequestDispatcher("/requestDataCenter.jsp");
    rd.forward(request, response);
}
if (action.equals("Requête"))
{
    RequestDispatcher rd = sc.getRequestDispatcher("/requestDataCenter.jsp");
    rd.forward(request, response);
}
if (action.equals("RequestHandling"))
{
    if (!isAuthenticated(request))
    {
        sc.log("-- Validation non terminée");
        RequestDispatcher rd = sc.getRequestDispatcher("/LoginDataCenter.jsp?msg=" +
            URLEncoder.encode("Veuillez d'abord terminer l'authentification !"));
        rd.forward(request, response);
    }
    else
    {
        sc.log("-- Traitement de la requête");
        RequestDispatcher rd = sc.getRequestDispatcher("/responseDataCenter.jsp");
        rd.forward(request, response);
    }
}
if (action.equals("NewRequest"))
{
    String reponse = request.getParameter("reponse");
    if (reponse.equals("Oui"))
    {
        sc.log("-- Vers une nouvelle requête");
        RequestDispatcher rd = sc.getRequestDispatcher("/requestDataCenter.jsp");
        rd.forward(request, response);
    }
    else
}

```

```

    {
        sc.log("-- Retour au login");
        RequestDispatcher rd = sc.getRequestDispatcher("/SimiliLoginDataCenter.jsp");
        rd.forward(request, response);
    }
}
out.close();
}

protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, java.io.IOException { processRequest(request, response); }

protected void doPost(HttpServletRequest request, HttpServletResponse response)
throws ServletException, java.io.IOException { processRequest(request, response); }

/** Returns a short description of the servlet.
 */
public String getServletInfo() { return "Servlet contrôleur"; }

private boolean isAuthenticated (HttpServletRequest request)
{
    HttpSession session = request.getSession(true);
    // Object existe = session.getValue("UserValid");
    Object existe = session.getAttribute("UserValid");
    return existe!=null;
}

private boolean isAuthenticationInProgress (HttpServletRequest request)
{
    HttpSession session = request.getSession(true);
    // Object existe = session.getValue("UserValid");
    Object existe = session.getAttribute("ValidationInProgress");
    return existe!=null;
}

private void setAuthenticated (HttpServletRequest request, boolean b)
{
    HttpSession session = request.getSession(true);
    if (b) session.setAttribute("UserValid", "Ok");
    else session.removeAttribute("UserValid");
}

private void setAuthenticationInProgress (HttpServletRequest request, boolean b)
{
    HttpSession session = request.getSession(true);
    if (b) session.setAttribute("ValidationInProgress", "Ok");
    else session.removeAttribute("ValidationInProgress");
}
}

```

15.7 Un exemple d'utilisation

Avec un serveur Tomcat sur une machine myriam (port 8080), cela pourrait donner :

Login sur le Data Center des références - Netscape

File Edit View Go Bookmarks Tools Window Help

http://myriam:8080/LoginDataCenter.jsp Search

Bienvenue sur LE site des références scientifiques !

Nous sommes le mercredi 4 août 2004 13 h 11 CEST !!!

Veuillez entrer votre nom et votre mot de passe ...

Authentification préalable, requête (après authentification obligatoire) ou fin ?"

Authentification ▾

Nom : vilvens

Mot de passe :

action

Document: Done (16.14 secs)

Authentification sur le Data Center (1) - Netscape

File Edit View Go Bookmarks Tools Window Help

http://myriam:8080/servlet/ControleDataCenter Search

Vous souhaitez donc entrer dans notre grande famille de chercheurs - félicitations !

Vous êtes donc : vilvens Confirmez votre mot de passe :

Mot de passe :

confirmer

Document: Done (0.171 secs)

Authentification sur le Data Center (2) - Netscape

File Edit View Go Bookmarks Tools Window Help

http://myriam:8080/servlet/ControleDataCenter

Vous êtes donc : vilvens avec le mot de passe : grosZZ

Nous souhaitons quelques renseignements complémentaires

Prénom :

Votre e-mail :

Document: Done (4.219 secs)

Requête au le Data Center (1) - Netscape

File Edit View Go Bookmarks Tools Window Help

http://myriam:8080/servlet/ControleDataCenter

Bonjour cher claude vilvens !

Je vous écoute : que cherchez-vous ?"

Auteur :

Année de publication :

La pensée du jour :

A vaincre sans péril, on triomphe sans gloire

Document: Done (6.516 secs)

Réponse du Data Center (1) - Netscape

File Edit View Go Bookmarks Tools Window Help

http://myriam:8080/servlet/ControleDataCenter Search

Mail Home Radio My Netscape Search Bookmarks

Voici les références correspondant à votre demande :

0 référence ... :-(

Autre requête ? Oui ▾

action

La pensée du jour :

A vaincre sans péril, on triomphe sans gloire

Document: Done (5.172 secs)

Data Center des références - Netscape

File Edit View Go Bookmarks Tools Window Help

http://myriam:8080/servlet/ControleDataCenter Search

Mail Home Radio My Netscape Search Bookmarks

Vous êtes toujours sur LE site des références scientifiques !

Nous sommes le mercredi 4 août 2004 17 h 56 CEST !!!

Requête (vous êtes déjà authentifié) ou fin ?"

Requête ▾

action

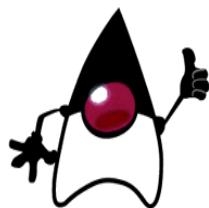
La pensée du jour :

A vaincre sans péril, on triomphe sans gloire

Document: Done (5.703 secs)

Le fichier de log contient ceci :

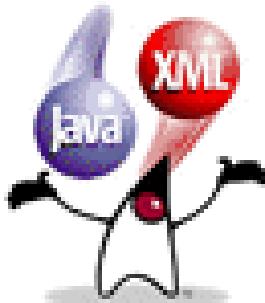
localhost_log.2004-08-04.txt
2004-08-04 17:48:47 -- démarrage de la servlet ControleDataCenter
2004-08-04 17:48:47 -- passage par la servlet ControleDataCenter
2004-08-04 17:48:47 -- Valeur du paramètre action : Authentification
2004-08-04 17:48:47 Nom récupéré par paramètre = vilvens -- ok ?
2004-08-04 17:48:48 -- Tentative de redirection sur AuthenticateDataCenter.jsp
2004-08-04 17:49:01 -- passage par la servlet ControleDataCenter
2004-08-04 17:49:01 -- Valeur du paramètre action : AuthenticationInProgress
2004-08-04 17:49:01 Nom récupéré = vilvens
2004-08-04 17:49:01 Nouveau mot de passe éventuel : grosZZ
2004-08-04 17:49:01 -- Tentative de redirection sur Authenticate2DataCenter.jsp
2004-08-04 17:49:22 -- passage par la servlet ControleDataCenter
2004-08-04 17:49:22 -- Valeur du paramètre action : AuthenticateRecorded
2004-08-04 17:49:22 -- Enregistrement du nouveau membre
2004-08-04 17:49:22 Nom récupéré = vilvens
2004-08-04 17:53:45 -- passage par la servlet ControleDataCenter
2004-08-04 17:53:45 -- Valeur du paramètre action : RequestHandling
2004-08-04 17:53:45 -- Traitement de la requête
2004-08-04 17:56:04 -- passage par la servlet ControleDataCenter
2004-08-04 17:56:04 -- Valeur du paramètre action : NewRequest
2004-08-04 17:56:04 -- Retour au login



Nous pourrions encore disserter sur ce passionnant sujet des applications Web sauce Java. Mais déjà d'autres sirènes de partage de l'information nous appellent, ceci du point de vue d'un formalisme commun des données échangées.

Par ici ...

XVIII. Les APIS Java pour XML



*Et je dis que le bonheur est inaccessible
Et je dis que ton cas n'est pas si désespéré
A condition d'analyser
L'étonnante précarité de nos amours
Destitués
;-)*

(Les Inconnus, "Tranxen 200")

XML (eXtensible Markup Language) s'est imposé ces dernières années comme un standard de la représentation des données structurées selon des règles précises et définies par des utilisateurs aux centres d'intérêt commun. En ce sens, on peut presque parler de "protocoles" de représentation et d'échange de données.

L'objectif étant, dans le présent ouvrage, de parler de la programmation Java de protocoles applicatifs, l'étude des packages permettant d'utiliser XML trouve donc sa place ici. Par contre, il n'entre pas dans ces objectifs de développer XML lui-même. Néanmoins, on trouvera en annexe, non pas un cours de XML, mais un bref exposé des notions de base nécessaires à la compréhension du présent chapitre.

XML étant donc supposé connu dans ses grandes lignes, attaquons-nous aux APIs Java correspondantes.

1. Les APIs XML

1.1 Les parsers XML

Le traitement des fichiers XML est du ressort des "**parsers XML**", soit des applications qui, lorsqu'elles sont complètes :

- ◆ vérifient la validité d'un fichier XML (par rapport à une **DTD - Document Type Definition/Declaration**);
- ◆ interprètent les informations contenues (parsing proprement dit) pour construire l'arborescence correspondante;
- ◆ transforment des informations en une entité manipulable par une autre application, comme un browser (au moyen d'APIs **XSLT - eXtensible Stylesheet Language Transformation**).

En ce qui concerne l'interprétation de documents XML, il existe à l'heure actuelle deux principaux APIs permettant de la réaliser : **SAX** et **DOM** – nous allons y revenir.

L'un des parsers les plus répandus est **Xerces**¹¹, produit par Apache Software. Il comporte des classes Java et C++ réalisant les opérations de parsing et de génération XML, avec le support des APIs SAX et DOM.

Apache produit également **Xalan**¹², processeur de feuilles de styles XML réalisant les transformations XSLT en se servant des feuilles de style qui lui sont fournies comme paramètres. Il implémente les recommandations XSLT et XPath de W3C.

¹¹ d'après le nom d'un papillon bleu du genre Xerces

¹² d'après le nom d'un instrument de musique

1.2 L'API SAX

SAX (Simple Api for Xml) est un ensemble d'APIs construit sur la notion d'événement. Il a été développé par les membres d'une mailing list (xml-dev) gérée par **OASIS** (Organization for Advancement of Structured Information Standards)¹³. SAX existe à l'heure actuelle en la version 2.0, qui ajoute la gestion des namespaces par rapport à la version 1.0.

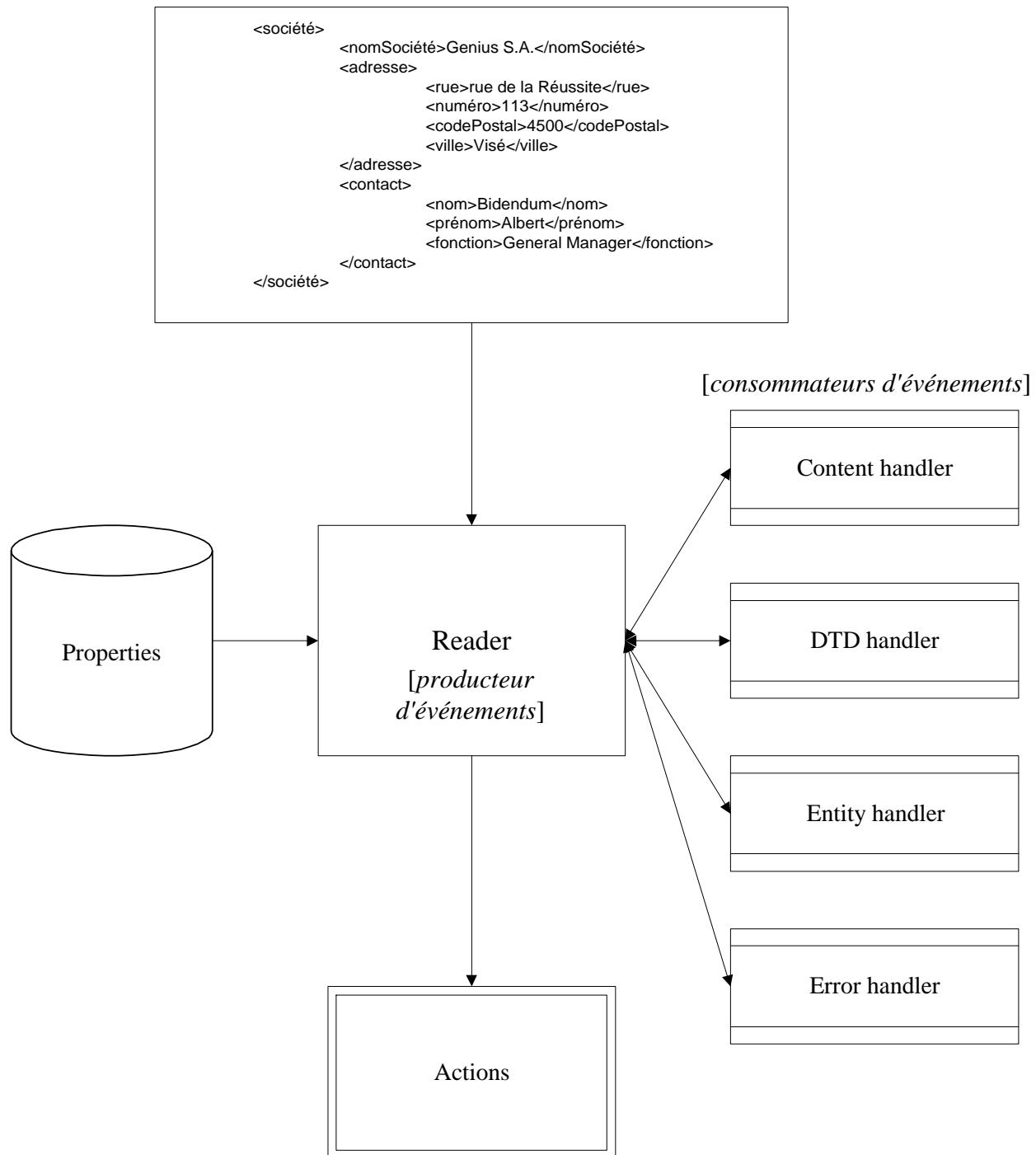
SAX est considéré comme étant de bas-niveau, c'est-à-dire qu'il réclame un travail de programmation plus grand que l'autre API DOM, mais en étant plus rapide et en utilisant moins de mémoire.

Concrètement, le document XML est lu séquentiellement et génère au fur et à mesure de la progression de cette lecture une suite d'événements du type : "début d'élément", "attribut d'un élément", "fin d'un élément", etc. En ce sens, on dit encore qu'un parser SAX est un **producteur d'événements**. De ce fait, un composant essentiel d'un parser SAX est ce que l'on appelle le **reader**; celui-ci génère divers événements qui seront "consommés" par des **handlers**, au sens événementiel du terme - les méthodes de ceux-ci sont donc des fonctions callback. L'ensemble de ces handlers constituent ce que l'on appelle un **consommateur d'événements**. On peut y distinguer :

- ◆ le Content Handler : il gère évidemment le contenu du document XML traité;
- ◆ le DTD Handler : comme son nom l'indique, il utilise la DTD détectée pour valider ou nom la suite du document;
- ◆ l>Error Handler : très clair ...
- ◆ l'Entity Resolver : il gère les entités extérieures (comme des paramètres par exemple) .

Schématiquement :

¹³ <http://www.oasis-open.org>



On remarquera que le reader peut être configuré selon des spécifications standard ou vendeurs.

On l'a dit, cette technique est considérée comme légère en ce sens qu'elle ne nécessite pas la présence de l'entièreté du document XML en mémoire (à la différence de l'autre API DOM, comme nous allons le voir). Cependant, revers de la médaille, ces APIs ne peuvent modifier un document ni naviguer aléatoirement dans le document.

1.3 L'API DOM

DOM (Document Object Model) est un ensemble d'APIs construit sur la notion d'arbre. Le document à analyser est représenté en mémoire sous la forme d'un arbre pour être ensuite traité, interrogé ou modifié (il peut même franchement être créé à partir de rien – *from scratch*). Revers de la médaille, *toutes les informations du document doit se trouver en mémoire*, ce qui implique une grosse consommation de ressources.

Les APIs DOM sont donc capables d'effectuer tous les traitements possibles sur le document XML considéré. Les spécifications de DOM sont établies dans des recommandations du W3C (niveau 1 10/1998 et niveau 2 11/2000 – le niveau 3 est en vue).

DOM se présente en fait sous forme d'interfaces. Ce sont donc les vendeurs qui fournissent les classes qui implémentent ces interfaces. Les APIs sont multi-langages, autrement dit indépendantes des langages de programmation et des systèmes d'exploitation. On ne s'étonnera donc pas que les interfaces DOM soient spécifiés en IDL (Interface Description Language) ...

2. XML et des packages en plus ...

Bien sûr, au niveau de Java, de nouveaux packages vont nous être nécessaires pour programmer en Java nos traitements XML. Il existe en fait différents APIs Java disponibles, dont les deux plus répandues sont :

- ◆ **JAXP (Java Api for Xml Processing)** : cette librairie permet de réaliser des parsings SAX et DOM ainsi que des transformations XSLT.
- ◆ **JDOM (Java DOM)** : proposée comme une synthèse des principes SAX et DOM, cette librairie permet une vue complète d'un document XML sans que ce document soit entièrement en mémoire. Elle permet également la création de documents à partir de rien ainsi que les déplacements aléatoires au sein de ceux-ci.

Pour notre part, nous utiliserons l'API **JAXP 1.1** qui couvre :

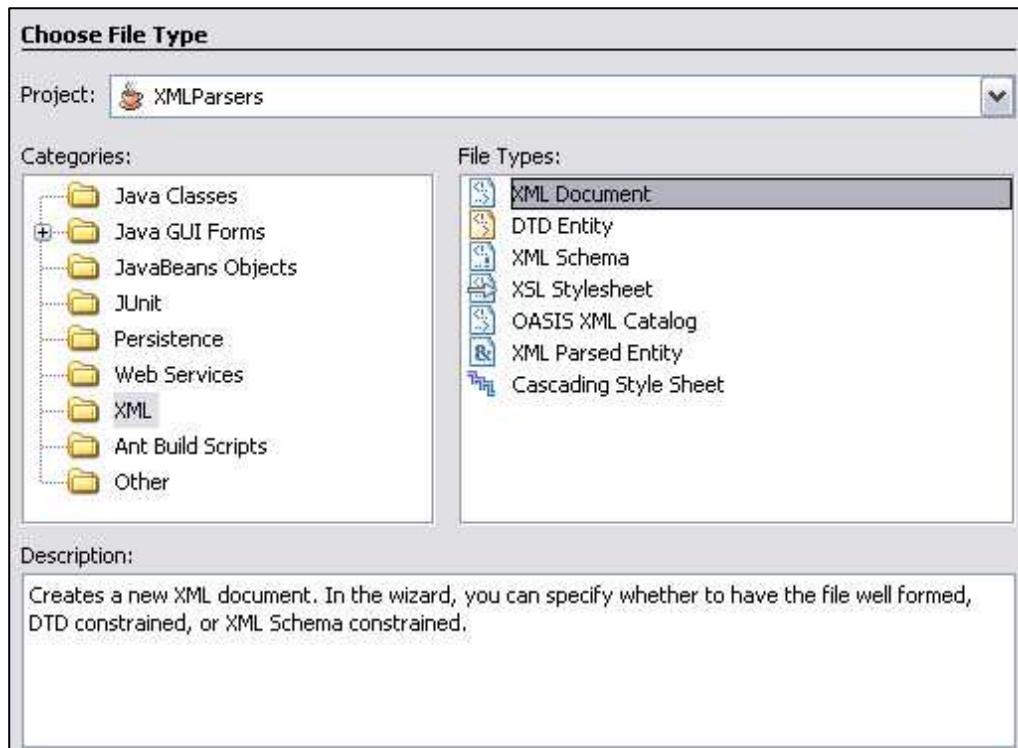
- ◆ XML 1.0 Second Edition;
- ◆ les Namespaces XML;
- ◆ SAX 2.0;
- ◆ SAX2 Extensions version 1.0;
- ◆ DOM selon la recommandation Level 2 Core Recommendation;
- ◆ XSLT 1.0.

Les **packages** correspondants, **intégrés au JDK depuis la version 1.4** mais aussi disponibles sur le site de Sun (dans jaxp.jar pour le deux premiers, dans crimson.jar pour les autres), sont :

javax.xml.parsers	: Les classes fondamentales qui représentent un parser SAX, un builder DOM et les factories correspondantes
javax.xml.transform	: Comme le nom l'indique, on trouve ici les APIs de transformation d'un document XML en une représentation donnée.
org.xml.sax	: Les interfaces reconnus de tous les acteurs SAX du traitement XML, comme les Reader, Content Handler, Error Handler, etc.
org.xml.sax.helpers	: Les classes de base par défaut, dont DefaultHandler qui est un adaptateur des interfaces cités dans le package précédent.
org.w3c.dom	: Les interfaces reconnus de tous les acteurs DOM du traitement XML, comme les Document, Element, etc

3. Les documents XML sous NetBeans

NetBeans permet de créer et gérer des documents XML avec beaucoup de facilité. Comme d'habitude, il suffit de choisir New → File/Folder, puis

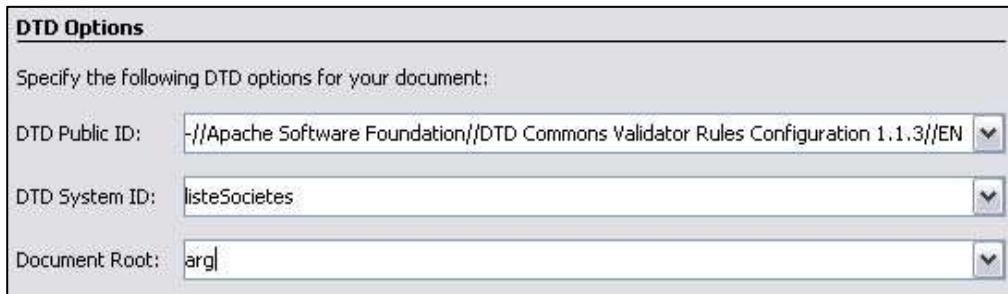


pour créer un document XML. La suite du Wizzard est logique :

The first screenshot shows the 'Name and Location' step of the wizard. It has fields for 'File Name' (set to 'listeSociétés'), 'Project' (set to 'XMLParsers'), 'Folder' (with a 'Browse...' button), and 'Created File' (showing the full path 'C:\java-netbeans-application\XMLParsers\listeSociétés.xml').

The second screenshot shows the 'Select Document Type' step. It contains a note: 'Select the type of XML document you want to create based on your document structure, data types, and namespace requirements.' Below this are three radio buttons: 'Well-formed Document' (selected), 'DTD-Constrained Document', and 'XML Schema-Constrained Document'.

Ici, un "well-formed document" est un simple fichier XML dont sera vérifié simplement le bien fondé des emboîtements des tags – si on choisit plutôt "DTD-Constrained Document", c'est-à-dire avec une DTD intégrée, on pourra désigner cette DTD si elle existe déjà :



Supposons que nous ayons créé de la même manière un fichier DTD :

listeSocietes.dtd

```
<?xml encoding="UTF-8" ?>

<!-- Created by Vilvens on 26 mai 2005, 9:05 -->

<!ELEMENT listeSocietes ANY>
<!ELEMENT societe (nomSociété, adresse, contact, numéroEnregistrement)>
<!ELEMENT nomSociété (#PCDATA)>
<!ELEMENT adresse (rue, numéro, codePostal, ville)>
<!ELEMENT rue (#PCDATA)>
<!ELEMENT numéro (#PCDATA)>
<!ELEMENT codePostal (#PCDATA)>
<!ELEMENT ville (#PCDATA)>
<!ELEMENT contact (nom, prénom, fonction, (telephone | email*)) >
<!ELEMENT nom (#PCDATA)>
<!ELEMENT prénom (#PCDATA)>
<!ELEMENT fonction (#PCDATA)>
<!ELEMENT email (#PCDATA)>
<!ELEMENT telephone (#PCDATA)>
<!ELEMENT numéroEnregistrement (#PCDATA)>
<!ATTLIST numéroEnregistrement ONSS ID #REQUIRED interne CDATA #IMPLIED>
```

pour ensuite compléter le fichier XML afin d'obtenir le résultat suivant (on remarquera la référence au fichier DTD sous forme d'URI, car le protocole Web – ici "file" – sera nécessaire pour les futurs parsers) :

listeSociétés.xml (version avec erreurs)

```
<?xml version="1.0" encoding="UTF-8"?>

<!--
Document : listeSociétés.xml
Created on : 26 novembre 2006, 12:25
Author : Vilvens
Description: Purpose of the document follows.
-->

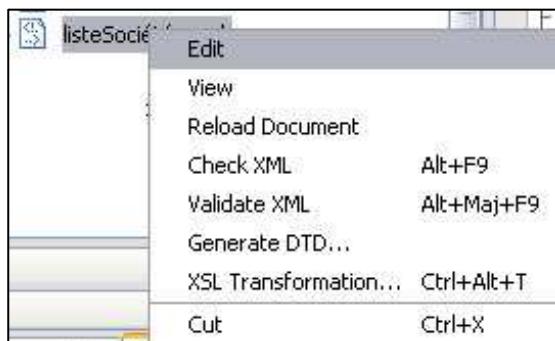
<!DOCTYPE listeSocietes SYSTEM 'file:/C:/java-netbeans-
application/XMLParsers/src/listeSocietes.dtd'>
```

```

<mesListesDeSociétés>
    <societe>
        <nomSociété>Genius S.A.</nomSociété>
        <adresse>
            <rue>rue de la Réussite</rue>
            <numéro>113</numéro>
            <codePostal>4500</codePostal>
            <ville>Visé</ville>
        </adresse>
        <contact>
            <nom>Bidendum</nom>
            <prénom>Albert</prénom>
            <fonction>General Manager</fonction>
            <email>abidend@groumf.com</email>
            <email>albert@sexy.net</email>
        </contact>
        <numéroEnregistrement ONSS="AG007" interne="BOND">501/213/40001
        </numéroEnregistrement>
    </societe>
    <societe>
        <nomSociété>Mes belles miches S.P.R.L.</nomSociété>
        <adresse>
            <rue>rue de la Boulangerie</rue>
            <numéro>69</numéro>
            <codePostal>4400</codePostal>
            <ville>Herstal</ville>
        </adresse>
        <contact>
            <nom>Brioche</nom>
            <prénom>Julie</prénom>
            <fonction>Maître</fonction>
            <email>julie.brioche@vilvens.net</email>
        </contact>
        <numéroEnregistrement ONSS="AG007"
            interne="FRANCESCA">502/4781/80001
        </numéroEnregistrement>
    </societe>
</mesListesDeSociétés>

```

Un clic droit sur le fichier xml permet de faire apparaître un menu contextuel :



grâce auquel on peut :

- ◆ éditer sous forme de texte (Edit);
- ◆ éditer en terme d'arborescence (View) – en fait, c'est un document visualisé dans un browser;
- ◆ vérifier la structure des tags (Check XML);
- ◆ valider le document par rapport à sa DTD – ici :

```
Checking file:/C:/java-netbeans-application/XMLParsers/src/listeSoci%C3%A9t%C3%A9s.xml...
Referenced entity at "file:/C:/java-netbeans-application/XMLParsers/src/listeSocietes.dtd".
Document root element "mesListesDeSociétés", must match DOCTYPE root "listeSocietes". [13]
Element type "mesListesDeSociétés" must be declared. [13]
Attribute value "AG007" of type ID must be unique within the document. [46]
XML validation finished.
```

Il semblerait donc que certaines fautes se soient glissées dans notre œuvre – corrigeons :

listeSociétés.xml (version validée)

```
<?xml version="1.0" encoding="UTF-8"?>

<!--
  Document : listeSociétés.xml
  Created on : 26 novembre 2006, 12:25
  Author   : Vilvens
  Description: Purpose of the document follows.
-->

<!DOCTYPE listeSocietes SYSTEM "listeSocietes.dtd">

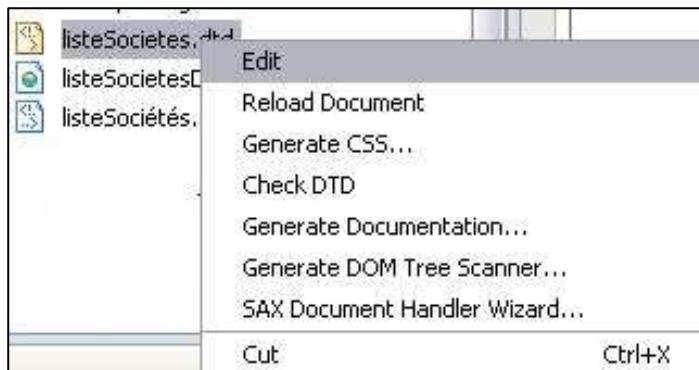
<listesSocietes>
    <societe> ... </societe>
    <societe>
        <nomSociété>Mes belles miches S.P.R.L.</nomSociété>
        ...
        <numéroEnregistrement ONSS="AG008"
            interne="FRANCESCA">502/4781/80001
        </numéroEnregistrement>
    </societe>
</listesSocietes>
```

- une nouvelle tentative nous montre que le document est en ordre /

```
XML validation started.
Checking file:/C:/java-netbeans-application/XMLParsers/src/listeSoci%C3%A9t%C3%A9s.xml...
Referenced entity at "file:/C:/java-netbeans-application/XMLParsers/src/listeSocietes.dtd".
XML validation finished.
```

- ◆ générer une DTD si on n'en a pas créé.

De manière similaire, un clic droit sur le fichier dtd permet de faire apparaître un menu contextuel :



Par exemple, le générateur de documentation fournit une page HTML qui rappelle les javadocs :

Element Index

- ◆ [adresse](#)
- ◆ [codePostal](#)
- ◆ [contact](#)
- ◆ [email](#)
- ◆ [fonction](#)
- ◆ [listeSocietes](#)
- ◆ [nom](#)
- ◆ [nomSociété](#)
- ◆ [numéro](#)
- ◆ [numéroEnregistrement](#)
- ◆ [prénom](#)
- ◆ [rue](#)
- ◆ [societe](#)
- ◆ [telephone](#)
- ◆ [ville](#)

Element Details

listeSocietes

contact

telephone

Element Content Model

(#PCDATA)*

Referenced by

[contact](#)

numéroEnregistrement

Declared Attributes

- ◆ #REQUIRED ID ONSS
- ◆ #IMPLIED CDATA interne

Element Content Model

(#PCDATA)*

Referenced by

[societe](#)

4. Un exemple de parser SAX

Pour fixer les idées, supposons vouloir parser le document XML ci-dessus pour compter ses tags (d'accord, on peut trouver plus intelligent – cela viendra bien un jour ;-) ...).

4.1 Le handler de base

NetBeans permet de générer un groupe de classes et interfaces pour les besoins du parsing – mais il s'agit d'une solution à profil propriétaire. Nous allons donc ici confectionner un parser depuis rien. Ou plutôt, depuis une classe de base appelée **DefaultHandler** (du package org.xml.sax.helpers). Celle-ci implémente les 4 interfaces qu'un parser SAX complet doit savoir faire gérer :

```
public class DefaultHandler  
implements EntityResolver, DTDHandler, ContentHandler, ErrorHandler  
{ ... }
```

Dans un premier temps, l'interface le plus important pour notre objectif est **ContentHandler** (du package org.xml.sax), dont les méthodes les plus utiles sont :

```
public void characters(char[] ch, int start, int length) throws SAXException  
    Reçoit la notification d'un caractère  
public void startDocument() throws SAXException  
    Reçoit la notification du début de document (en principe, cette méthode n'est donc appellée qu'une fois)  
public void endDocument() throws SAXException  
    Reçoit la notification de la fin de document (cette méthode n'est donc appelée qu'une fois, à la fin effective ou en cas d'erreur)  
public void startElement(java.lang.String namespaceURI, java.lang.String localName,  
    java.lang.String qName, Attributes atts) throws SAXException  
    Reçoit la notification du début d'un tag : on constate que le tag est décomposé en préfixe (c'est-à-dire le nom du namespace), nom local, nom qualifié (c'est-à-dire le nom complet) et attributs.  
public void endElement(java.lang.String namespaceURI, java.lang.String localName,  
    java.lang.String qName) throws SAXException  
    Reçoit la notification de fin d'un tag.
```

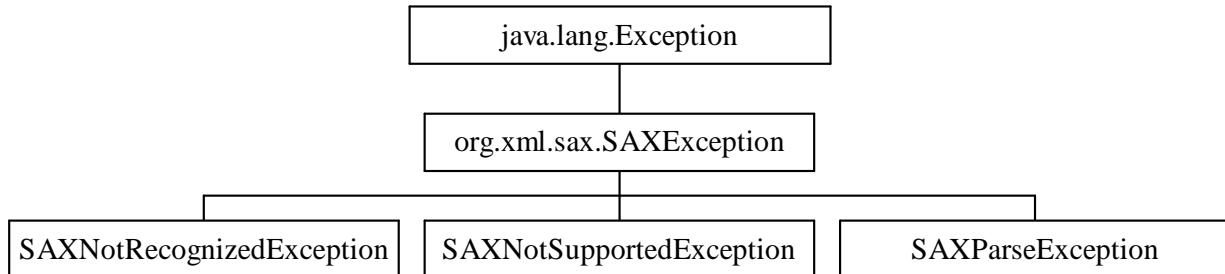
Les attributs éventuels sont donc récupérés dans un objet qui implémente l'interface **Attributes** (du package org.xml.sax). Quelques méthodes utiles sont :

```
public int getLength()  
    Fournit le nombre d'attributs  
public java.lang.String getLocalName(int index)  
    Fournit le nom d'un attribut selon sa position  
public java.lang.String getType(int index)  
    Fournit le nom d'un attribut selon sa position; les types possibles sont : "CDATA",  
    "ID", "IDREF", "IDREFS", "NMTOKEN", "NMTOKENS", "ENTITY", "ENTITIES", or  
    "NOTATION".  
public java.lang.String getValue(int index)  
    Fournit la valeur d'un attribut selon sa position
```

Evidemment, les implémentations des diverses méthodes sont minimales (elles ne font rien) et il nous appartiendra donc de redéfinir celles qui nous préoccupent.

4.2 L'exception de SAX

On s'en doute, la classe d'exception **SAXException** sert à référencer les erreurs concernant les traitement XML selon l'API SAX. Dérivée de Exception (de java.lang), cette classe SAXException fait partie du package org.xml.sax. Elle possède trois filles :



4.3 Obtenir un parser

Le parser proprement dit sera une instance dérivée de la classe abstraite **SAXParser** (du package javax.xml.parsers), ou plutôt une espèce d'implémentation puisque cette classe est abstraite. Elle encapsule un objet qui implémente l'interface **XMLReader**. Ce dernier prototype une armée de méthodes parse() polymorphes, comme :

```
public void parse(java.io.File f, DefaultHandler dh)
    throws SAXException, java.io.IOException
```

qui parse effectivement le fichier XML précisé *au moyen du handler précisé*. Inutile de dire que le parser lui-même propose (afin de ne pas devoir systématiquement passer par l'objet XMLReader encapsulé) des méthodes au prototype semblable :

```
public void parse(java.io.File f, DefaultHandler dh)
    throws SAXException, java.io.IOException
```

Pour instancier le parser lui-même, il nous faudra obtenir une instance de SAXParser au moyen d'une factory. En fait, nous allons utiliser une *classe factory*, **SAXParserFactory** (du package javax.xml.parsers) *au lieu d'une méthode factory* : la différence est que l'on peut ainsi regrouper en une seule entité non seulement la méthode factory proprement dite mais les méthodes annexes de configuration, de manière à finalement obtenir un parser SAX répondant à nos besoins.

1) La première méthode factory à utiliser est :

```
public static SAXParserFactory newInstance()
```

qui permet d'obtenir une instance de SAXParserFactory *indépendamment de la manière dont cette instance est effectivement implémentée*. En fait, cette méthode de classe va successivement rechercher la classe implémentant la factory

- ◆ dans la propriété du système nommée javax.xml.parsers.SAXParserFactory;
- ◆ dans le fichier lib/jaxp.properties du JRE;
- ◆ dans les API de service (Services API) qui sont éventuellement définies dans META-INF/services/javax.xml.parsers.SAXParserFactory (c'est le cas dans crimson.jar);

- ♦ dans l'instance de SAXParserFactory par défaut pour le système.

Donc, finalement :

SAXParserFactory *spf* = SAXParserFactory.newInstance();

2) Cette classe factory va maintenant nous permettre d'obtenir une instance de parser. Auparavant, on peut configurer la factory avec des méthodes comme :

```
public void setNamespaceAware (boolean awareness)
public void setValidating (boolean validating)
```

au rôle bien clair. Le parser s'obtient en utilisant la méthode de classe

```
public abstract SAXParser newSAXParser()
    throws ParserConfigurationException, SAXException
```

qui utilise les valeurs des paramètres courants [*features*], éventuellement modifiés par d'autres méthodes de SAXParserFactory (comme nous n'avons rien fait, ce seront les valeurs par défaut). Donc :

SAXParser *sp* = *spf*.newSAXParser();

4.4 Les méthodes du ContentHandler

Les méthodes du handler par défaut ne font rien : elles se contentent d'exister. Il nous appartient donc de réécrire celles qui nous intéressent, ce que nous allons faire de la manière la plus simple possible. Ainsi, par exemple (la méthode trace() ne dissimule qu'un affichage console) :

```
public void startElement(java.lang.String uri, java.lang.String localName,
    java.lang.String qName, Attributes attr) throws SAXException
{
    trace("* Début d'un élément");
    cptTags++;
    trace("++ compteur de tags", cptTags);
    if (uri != null && uri.length()>0) trace("  uri", uri);
    trace("  nom local", localName);

    if (uri != null && uri.length()>0) trace("  nom complet", qName);
    int nAttr = attr.getLength();
    trace("  nombre d'attributs", nAttr);
    if (nAttr ==0) return; // Denys like
    for (int i=0; i<nAttr; i++)
        trace("    attribut n°" + i + " = " + attr.getLocalName(i) +
            " avec valeur : " + attr.getValue(i));
}

public void endElement(java.lang.String uri, java.lang.String localName,
    java.lang.String qName) throws SAXException
```

```
{
    trace("* Fin de l'élément " + localName);
    cptTags++;
    trace("++ compteur de tags", cptTags);
}
```

4.5 Le parser minimum

En regroupant toutes nos réflexions, nous pouvons écrire un parser compteur de tags de la manière suivante :

SaxCompteur.java

```
/*
 * SaxCompteur.java
 */
/**
 * @author Vilvens
 */

package parsers;

import javax.xml.parsers.*;
import org.xml.sax.*;
import org.xml.sax.helpers.*;
import java.io.*;

public class SaxCompteur extends DefaultHandler
{
    protected String nomFichierXML;
    int cptTags =0;

    public SaxCompteur()
    {
        setNomFichierXML(null);
    }
    public SaxCompteur(String nfx)
    {
        setNomFichierXML(nfx);
    }

    public String getNomFichierXML() { return nomFichierXML; }
    public void setNomFichierXML(String nfx) { nomFichierXML=nfx; }

    static protected void trace (String s)
    {
        System.out.println(s);
    }
    static protected void trace (String sCte, String s)
    {
        System.out.println(sCte + " : " + s);
    }
}
```

```

static protected void trace (String sCte, int i)
{
    System.out.println(sCte + " : " + i);
}

// Quelques méthodes du ContentHandler
public void characters(char[] ch, int start,int length) throws SAXException
{
    String chaine = new String(ch, start, length).trim();
    if (chaine.length() > 0) trace("@ Caractères", chaine);
}

public void startDocument()throws SAXException
{
    trace("## Début du document **");
}

public void endDocument()throws SAXException
{
    trace("## Fin du document **");
}

public void startElement(java.lang.String uri, java.lang.String localName,
                      java.lang.String qName, Attributes attr) throws SAXException
{
    trace("* Début d'un élément");
    cptTags++;
    trace("++ compteur de tags", cptTags);
    if (uri != null && uri.length()>0) trace("    uri", uri);
    trace("    nom complet", qName);
    if (uri != null && uri.length()>0) trace("    nom complet", qName);
    int nAttr = attr.getLength();
    trace("    nombre d'attributs", nAttr);
    if (nAttr ==0) return; // Denys like

    for (int i=0; i<nAttr; i++)
        trace("        attribut n°" + i + " = " + attr.getLocalName(i) +
              " avec valeur : " + attr.getValue(i));
}

public void endElement(java.lang.String uri, java.lang.String localName,
                      java.lang.String qName) throws SAXException
{
    trace("* Fin de l'élément " + localName);
    cptTags++;
    trace("++ compteur de tags", cptTags);
}

```

```

public static void main (String args[])
{
    trace("Création du handler");
    SaxCompteur sc =
        new SaxCompteur("c:\\java-netbeans-application\\XMLParsers\\src\\listeSociétés.xml");
    trace("Création de la factory");
    SAXParserFactory spf = SAXParserFactory.newInstance();

    try
    {
        trace("Création du parser");
        SAXParser sp = spf.newSAXParser();
        trace("Mise en route du parsing \\n\\n");
        sp.parse(new File(sc.getNomFichierXML()), sc);
    }
    catch (ParserConfigurationException e)
    {
        System.out.println("Oh oh Problème de config : " + e.getMessage());
    }
    catch (SAXException e)
    {
        System.out.println("Oh oh Problème de SAX : " + e.getMessage());
    }
    catch (IOException e)
    {
        System.out.println("Oh oh Problème d'IO : " + e.getMessage());
    }
}
}

```

Le résultat, appliqué au fichier listeSociétés.xml évoqué ci-dessus (mais restreint à la première société), ressemble à ceci :

Création du handler
 Création de la factory
 Création du parser
 Mise en route du parsing
**** Début du document ****
 * Début d'un élément
 ++ compteur de tags : 1
 nom complet : listeSociétés
 nombre d'attributs : 0
*** Début d'un élément**
 ++ compteur de tags : 2
 nom complet : société
 nombre d'attributs : 0

*** Début d'un élément**

++ compteur de tags : 3
nom complet : nomSociété
nombre d'attributs : 0

@ Caractères : Genius S.A.

*** Fin de l'élément nomSociété**

++ compteur de tags : 4

*** Début d'un élément**

++ compteur de tags : 5
nom complet : *adresse*
nombre d'attributs : 0

*** Début d'un élément**

++ compteur de tags : 6
nom complet : rue
nombre d'attributs : 0

@ Caractères : rue de la Réussite

*** Fin de l'élément rue**

++ compteur de tags : 7

*** Début d'un élément**

++ compteur de tags : 8
nom complet : numéro
nombre d'attributs : 0

@ Caractères : 113

*** Fin de l'élément numéro**

++ compteur de tags : 9

*** Début d'un élément**

++ compteur de tags : 10
nom complet : codePostal
nombre d'attributs : 0

@ Caractères : 4500

*** Fin de l'élément codePostal**

++ compteur de tags : 11

*** Début d'un élément**

++ compteur de tags : 12
nom complet : ville
nombre d'attributs : 0

@ Caractères : Visé

*** Fin de l'élément ville**

++ compteur de tags : 13

*** Fin de l'élément adresse**

++ compteur de tags : 14

*** Début d'un élément**

++ compteur de tags : 15
nom complet : *contact*
nombre d'attributs : 0

*** Début d'un élément**

++ compteur de tags : 16
nom complet : nom
nombre d'attributs : 0

@ Caractères : Bidendum

```
* Fin de l'élément nom
++ compteur de tags : 17
* Début d'un élément
++ compteur de tags : 18
    nom complet : prénom
    nombre d'attributs : 0
@ Caractères : Albert
* Fin de l'élément prénom
++ compteur de tags : 19
* Début d'un élément
++ compteur de tags : 20
    nom complet : fonction
    nombre d'attributs : 0
@ Caractères : General Manager
* Fin de l'élément fonction
++ compteur de tags : 21
* Fin de l'élément contact
++ compteur de tags : 22
* Fin de l'élément société
++ compteur de tags : 23
* Fin de l'élément listeSociétés
++ compteur de tags : 24
** Fin du document **
```

Evidemment, ce n'est pas très excitant ! Visons donc plus haut ;-) ...

5. Un parser qui utilise la réflexion : le principe du projet "robots"

Afin d'illustrer l'utilisation "industrielle" d'un parsing comme celui évoqué dans le paragraphe précédent, considérons le problème classique du pilotage d'un robot mobile. Celui-ci est sensé capable d'effectuer des opérations élémentaires comme avancer d'une certaine distance ou tourner d'un certain angle. Certains modèles sont capable d'actions additionnelles, comme par exemple poser ou relever un crayon qui, si il est effectivement en contact avec le sol, laisse une trace derrière lui .

L'idée est de programmer ce mobile (qui n'est ici que simulé – 'faut pas pousser' ;-)) au moyen d'un fichier XML. Celui-ci comporte un certain nombre d'opérations élémentaires qui suivent une syntaxe du type

<nom> [<paramètre éventuel de distance ou de rotation>]

Un fichier XML de ce type pourrait être :

listeOpérations.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<!-- Created by Vilvens on 1 octobre 2002, 15:36 -->
```

```
<!DOCTYPE listeOpérations [  
    <!ELEMENT listeOpérations ANY>  
    <!ELEMENT opération (nom, (angle | distance)? )>  
    <!ELEMENT nom (#PCDATA)>  
    <!ELEMENT angle (#PCDATA)>  
    <!ELEMENT distance (#PCDATA)>  
>  
  
<listeOpérations>  
    <opération>  
        <nom>démarre</nom>  
    </opération>  
    <opération>  
        <nom>poseCrayon</nom>  
    </opération>  
    <opération>  
        <nom>tourne</nom>  
        <angle>30</angle>  
    </opération>  
    <opération>  
        <nom>avance</nom>  
        <distance>50</distance>  
    </opération>  
    <opération>  
        <nom>tourne</nom>  
        <angle>60</angle>  
    </opération>  
    <opération>  
        <nom>avance</nom>  
        <distance>80</distance>  
    </opération>  
    <opération>  
        <nom>lèveCrayon</nom>  
    </opération>  
    <opération>  
        <nom>monte</nom>  
    </opération>  
    <opération>  
        <nom>stoppe</nom>  
    </opération>  
</listeOpérations>
```

On peut donc effectivement parler de "script" de pilotage de robot, facilement réalisable par un utilisateur non informaticien (ou si peu).

Parser un tel fichier XML ne présente pas de problème pour nous. Nous pourrons ainsi assez facilement repérer les opérations à effectuer. Mais qui va les effectuer ? Réponse : un objet Java dont les méthodes portent le même nom que les opérations. Mais comment faire comprendre, avec souplesse, qu'une chaîne de caractères est en fait le nom d'une méthode à exécuter ? En utilisant l'API Reflection ...

6. L'introspection : découvrir les propriétés d'une classe

6.1 L'API Reflection

Pour chaque classe instanciée, l'environnement d'exécution de Java (le **JRE** – Java Runtime Environment) crée *un objet particulier dont le rôle est de contenir toutes les informations utiles sur cette classe*. Un tel objet instancie la classe ... **Class**, du package `java.lang` (nous y avons déjà fait allusion dans le chapitre consacré à JDBC). Les concepteurs de Java disent encore d'un tel objet qu'il "reflète" la classe examinée et les méthodes qui permettent d'obtenir les informations contenues dans l'objet Class constituent ce que l'on appelle le "Reflection API".

Quel est l'usage de cet API ? Evidemment, permettre à une application de découvrir les propriétés (les méthodes, les variables membres, les interfaces implémentées, ...) d'une classe évoquée **dynamiquement**, c'est-à-dire seulement connue (par son nom, par un objet qui l'instancie, ...) au moment de l'exécution parce qu'elle serait, par exemple, saisie au clavier ou sur un flux réseau. Il est un fait que les objets acquis dynamiquement, comme par exemple les objets "proxys" que nous allons évoquer plus loin dans les paragraphes consacrés à RMI, implémentent un interface connu de l'application utilisatrice; mais un tel objet peut très bien posséder des méthodes qui ne font pas partie de l'interface ...

6.2 S'informer sur la nature d'un objet

Si l'on dispose déjà d'un objet "venu d'ailleurs" et qui implémente peut-être un interface connu, on peut obtenir un objet **Class** qui nous renseignera sur la classe qu'il instancie effectivement au moyen de la méthode héritée d'**Object** :

```
public final native Class getClass()
```

Au sein de **Class**, la méthode

```
public native String getName()
```

permet d'obtenir le nom de cette classe. Donc, par exemple :

```
String msg = "Bonjour le monde";
Class quiEstTu = msg.getClass();
System.out.println("msg instancie " + quiEstTu.getName());
```

donne comme résultat :

```
| msg instancie java.lang.String
```

6.3 S'informer sur les caractéristiques d'une classe

A l'inverse, il se peut qu'une classe ne soit connue *que par son nom fourni seulement au moment de l'exécution*. Dans ce cas, les méthodes d'introspection vous nous permettre d'en savoir plus sur elle au cours du déroulement de l'application. Ainsi, pour fixer les idées, supposons que nous ayons construit une classe de nombres complexes :

classe nombreComplexe
class nombreComplexe
{
private double pR;

```

private double pI;

public nombreComplexe()
{
    pR=pI=0;
}
public nombreComplexe( double r, double i)
{
    pR=r; pI=i;
}

public void affiche()
{
    System.out.println("(+"+pR+";"+pI+ ")");
}
public void affiche(PrintStream ps)
{
    ps.println("(+"+pR+";"+pI+ ")");
}

public double getReel() { return pR; }
public double getImaginaire() { return pI; }
public void setReel(double v) { pR=v; }
public void setImaginaire(double v) { pI=v; }

public nombreComplexe addComplexe (nombreComplexe nc2)
{
    nombreComplexe res = new nombreComplexe();
    res.pR = pR + nc2.pR; res.pI = pI + nc2.pI;
    return res;
}

static public nombreComplexe sousComplexe (nombreComplexe nc1, nombreComplexe nc2)
{
    nombreComplexe res = new nombreComplexe();
    res.pR = nc1.pR - nc2.pR;
    res.pI = nc1.pI - nc2.pI;
    return res;
}
public String toString()
{
    String ch = "(+"+pR+";"+pI+ ")";
    return ch;
}
}

```

Pour découvrir le contenu de cette classe, qui ferait normalement partie d'un package quelconque et dont le nom serait fourni dynamiquement, il faut simplement demander au système de localiser cette classe et de la charger en mémoire. C'est ce que fait la méthode de classe de la classe Class (vous me suivez ;-) ?) :

```
public static native Class forName(String className) throws ClassNotFoundException
```

qui renvoie un objet Class en cas de succès et qui lance l'exception ClassNotFoundException en cas d'échec. En fait, cette méthode agit comme le ClassLoader. Celui-ci recherche le bytecode de la classe demandée et le "lie" à l'environnement d'exécution de la machine virtuelle. Ceci signifie que la cohérence de la classe est vérifiée (*verification*), que les membres statiques (c'est-à-dire de classe) sont mis en place (*preparation*) et que les références symboliques sont remplacées par de véritables références; il s'agit donc finalement d'une opération du type "**édition de liens**" (*linking*). Immédiatement après, les codes d'initialisations des membres statiques sont exécutés (*initialization*). A ce stade, aucun objet n'a donc été initialisé, mais ces composantes statiques sont disponibles !

L'objet Class récupéré après l'appel de forName est encore appelé très justement un "**Runtime Class Descriptor**". Ce que l'on peut en tirer, à titre informationnel, est donné par les méthodes de la classe Class. Citons d'emblée :

```
public native boolean isInterface()
```

qui permet de vérifier si l'on a affaire à une classe ou à un interface. Il ne s'agit pas ici de prétendre à l'exhaustivité; signalons donc principalement une méthode qui permet d'obtenir la liste des méthodes de la classe (pas les méthodes héritées) :

```
public Method[] getDeclaredMethods() throws SecurityException
```

On obtient en plus les méthodes héritées avec :

```
public Method[] getMethods() throws SecurityException
```

Ce que l'on récupère est donc en fait un tableau d'objets **Method**, classe déclarée dans le package java.lang.reflect et dont le rôle est de contenir des informations sur une méthode, comme par exemple son nom donné par :

```
public String getName()
```

Les constructeurs peuvent être obtenus d'une manière similaire au moyen de la méthode

```
public Constructor[] getDeclaredConstructors() throws SecurityException
```

la classe **Constructor** étant bien sûr déclarée dans le package java.lang.reflect et destinée à contenir des informations sur un constructeur.

En pratique, on peut donc programmer ceci :

complexes.java (partie informations)

```
import java.io.*;
import java.lang.reflect.*; // seul java.lang est importé d'office

class nombreComplexe
{ private double pR; ... }
```

```

public class complexes
{
    public complexes () {}
    public static void main(String args[])
    {
        String msg = "Bonjour le monde";
        Class quiEsTu = msg.getClass();
        System.out.println("msg instancie "+ quiEsTu.getName()); System.out.println(msg);

        Class info=null;
        try
        {
            info = Class.forName("nombreComplexe");
        }
        catch (ClassNotFoundException e)
        {
            System.out.println("Bof !? " + e.getMessage());
        }
        if (info==null) System.exit(0);

        Method infoMéthodes[] = info.getDeclaredMethods();
        for (int i=0; i<infoMéthodes.length; i++)
        {
            System.out.println("méthode["+i+"] = "+infoMéthodes[i]);
        }
        Constructor infoConstructeurs[] = info.getDeclaredConstructors();
        for (int i=0; i<infoConstructeurs.length; i++)
        {
            System.out.println("constructeur["+i+"] = "+infoConstructeurs[i]);
        }
        ... // la suite dans quelques instants ;-
    }
}

```

Cela donne :

```

msg instancie java.lang.String
Bonjour le monde
méthode[0] = public void nombreComplexe.affiche()
méthode[1] = public void nombreComplexe.affiche(java.io.PrintStream)
méthode[2] = public double nombreComplexe.getReel()
méthode[3] = public double nombreComplexe.getImaginaire()
méthode[4] = public void nombreComplexe.setReel(double)
méthode[5] = public void nombreComplexe.setImaginaire(double)
méthode[6] = public nombreComplexe nombreComplexe.addComplexe(nombreComplexe)
méthode[7] = public static nombreComplexe nombreComplexe.sousComplexe
(nombreComplexe, nombreComplexe)
méthode[8] = public java.lang.String nombreComplexe.toString()
constructeur[0] = public nombreComplexe()
constructeur[1] = public nombreComplexe(double,double)

```

6.4 L'instanciation et l'utilisation d'objets

La programmation de l'instanciation d'un objet suppose l'appel de l'un de ses constructeurs, lequel porte forcément le nom de la classe, nom que l'on ne connaîtra qu'au moment de l'exécution ... La méthode de Class :

```
public native Object newInstance() throws InstantiationException, IllegalAccessException
```

permet d'instancier un objet de la classe découverte par appel à son constructeur par défaut (celui-ci doit donc exister). Par exemple :

```
try
{
    ncn = (nombreComplexe)info.newInstance();
}
catch (InstantiationException e) {}
catch(IllegalAccessException e) {}
```

Il existe des moyens de faire référence aux constructeurs découverts par `getConstructors()`. Mais, plus globalement, il est possible d'invoquer les méthodes découvertes au moyen de la méthode de la classe Method :

```
public native Object invoke(Object obj, Object args[])
throws IllegalAccessException, IllegalArgumentException, InvocationTargetException
```

le premier paramètre représentant l'objet qui appelle la méthode (null si il s'agit d'une méthode de classe) et le second fournissant la liste des paramètres effectifs de cette méthode. On peut donc imaginer :

```
try
{
    infoMéthodes[0].invoke(ncn, null);
}
catch (IllegalAccessException e) {}
catch (IllegalArgumentException e) {}
catch (InvocationTargetException e) {}
```

ce qui appelle bien la méthode `affiche()` qui n'a pas de paramètres. Comment, précisément, savoir quels sont les paramètres d'une méthode donnée ? Au moyen de la méthode de la classe Method :

```
public Class[] getParameterTypes()
```

De même, il est possible de récupérer le type de la valeur de retour d'une méthode au moyen de :

```
public Class getReturnType()
```

On peut donc imaginer une suite (guère recherchée ...) au programme précédent :

complexes.java (partie instantiation et invocations)

```

import java.io.*;
import java.lang.reflect.*; // seul java.lang est importé d'office

class nombreComplexe { private double pR; ... }

public class complexes
{
    public complexes () {}
    public static void main(String args[])
    {
        ...
        nombreComplexe ncn;
        try
        {
            ncn = (nombreComplexe)info.newInstance();
        }
        catch (InstantiationException e) {}
        catch(IllegalAccessException e) {}
        ncn.affiche();
        ncn.setReel(78e-2); ncn.setImaginaire(-54e-2);
        ncn.affiche();
        try
        {
            infoMéthodes[0].invoke(ncn, null);
        }
        catch (IllegalAccessException e) {}
        catch (IllegalArgumentException e) {}
        catch (InvocationTargetException e) {}
    }
}

```

```

Class infoParamètres[] = infoMéthodes[6].getParameterTypes();
Object paramètres[] = new Object[infoParamètres.length];
paramètres[0] = new nombreComplexe(2,3);
nombreComplexe res=null;
try
{
    res = (nombreComplexe)infoMéthodes[6].invoke(ncn, paramètres);
}
catch (IllegalAccessException e) {}
catch (IllegalArgumentException e) {}
catch (InvocationTargetException e) {}
res.affiche();

// si on ne connaît pas le type de la valeur de retour
Class infoRetour = infoMéthodes[6].getReturnType();
if (infoRetour.getName().equals("void"))
{
    System.out.println("Pas de valeur de retour");
}

```

```

else
{
    try
    {
        Object valeurRetour;
        valeurRetour = infoMéthodes[6].invoke(ncn, paramètres);
        System.out.println("Valeur de retour = " + valeurRetour + "("
            + infoRetour.getName() + ")");
        // OK si la méthode toString a été redéfinie
    }
    catch (IllegalAccessException e) {}
    catch (IllegalArgumentException e) {}
    catch (InvocationTargetException e) {}
}
}
}

```

On obtient ainsi :

```
(0.0;0.0)
(0.78;-0.54)
(0.78;-0.54)
(2.780000000000002;2.46)
Valeur de retour = (2.780000000000002;2.46)(nombreComplexe)
```

6.5 Application aux drivers JDBC

On se souviendra qu'un driver JDBC est invoqué par un simple appel du type :

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

On l'aura à présent bien compris, ceci n'instancie pas un objet driver, mais met par contre en place les composantes statiques de la classe. Et c'est précisément ce qui importe, puisque la recherche des méthodes de la classe :

informations pour un driver JDBC

```

...
Class infoJDBC=null;
try
{
    infoJDBC = Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
}
catch (ClassNotFoundException e)
{
    System.out.println("Bof !? " + e.getMessage());
}
Method infoMéthodesJDBC[] = infoJDBC.getDeclaredMethods();
for (int i=0; i<infoMéthodesJDBC.length; i++)
{
    System.out.println("méthode["+i+"] = "+infoMéthodesJDBC[i]);
}
...

```

montre qu'un certain nombre de méthodes sont effectivement statiques !

```
méthode[0] = protected synchronized void sun.jdbc.odbc.JdbcOdbcDriver.finalize()
méthode[1] = public synchronized java.sql.Connection
sun.jdbc.odbc.JdbcOdbcDriver.connect(java.lang.String,java.util.Properties) throws
java.sql.SQLException
méthode[2] = public boolean sun.jdbc.odbc.JdbcOdbcDriver.acceptsURL(java.lang.String)
throws java.sql.SQLException
méthode[3] = public java.sql.DriverPropertyInfo[]
sun.jdbc.odbc.JdbcOdbcDriver.getPropertyInfo(java.lang.String,java.util.Properties) throws
java.sql.SQLException
méthode[4] = public int sun.jdbc.odbc.JdbcOdbcDriver.getMajorVersion()
méthode[5] = public int sun.jdbc.odbc.JdbcOdbcDriver.getMinorVersion()
méthode[6] = public boolean sun.jdbc.odbc.JdbcOdbcDriver.jdbcCompliant()
méthode[7] = private boolean sun.jdbc.odbc.JdbcOdbcDriver.initialize() throws
java.sql.SQLException
méthode[8] = private boolean sun.jdbc.odbc.JdbcOdbcDriver_knownURL(java.lang.String)
méthode[9] = public static java.lang.String
sun.jdbc.odbc.JdbcOdbcDriver.getProtocol(java.lang.String)
méthode[10] = public static java.lang.String
sun.jdbc.odbc.JdbcOdbcDriver.getSubProtocol(java.lang.String)
méthode[11] = public static java.lang.String
sun.jdbc.odbc.JdbcOdbcDriver.getSubName(java.lang.String)
méthode[12] = private boolean sun.jdbc.odbc.JdbcOdbcDriver_trusted()
méthode[13] = public java.lang.String
sun.jdbc.odbc.JdbcOdbcDriver.getConnectionAttributes(java.lang.String,java.lang.String)
throws java.sql.SQLException
méthode[14] = public java.util.Hashtable
sun.jdbc.odbc.JdbcOdbcDriver.getAttributeProperties(java.lang.String)
méthode[15] = protected static java.lang.String
sun.jdbc.odbc.JdbcOdbcDriver.makeConnectionString(java.util.Properties)
méthode[16] = protected static java.lang.String[]
sun.jdbc.odbc.JdbcOdbcDriver.listToArray(java.lang.String)
méthode[17] = public int sun.jdbc.odbc.JdbcOdbcDriver.allocConnection(int) throws
java.sql.SQLException
méthode[18] = public void sun.jdbc.odbc.JdbcOdbcDriver.closeConnection(int) throws
java.sql.SQLException
méthode[19] = public void sun.jdbc.odbc.JdbcOdbcDriver.disconnect(int) throws
java.sql.SQLException
```

Mais d'où pourraient venir des objets inconnus ? D'un "serveur d'objets" ? Nous en reparlerons. Revenons à nos robots ...

7. Un parser qui utilise la réflexion : la réalisation du projet "robots"

7.1 Les outils

Avec l'API Reflection et surtout la méthode :

```
public Object invoke(Object obj, Object[] args)
```

tous les espoirs sont permis, puisque un String lu dans le fichier XML est un Object ...

De plus, on peut imaginer qu'il existe différents fabricants de robots, ces robots ayant des possibilités similaires mais avec des spécificités et hébergeant des classes Java ayant un air de famille mais différentes par certains détails. Ceci nous fait penser à des notions classiques :

- ◆ les interfaces;
- ◆ les classes abstraites;
- ◆ les méthodes **factory**.

Avant que le vertige ne nous terrasse, voyons cela de plus près ...

7.2 Les classes robots

Vu la généralité de nos objectifs, il paraît logique de définir un interface fixant les possibilités minimales des objets robots. On peut donc imaginer :

mobile.java

```
/*
 * mobile.java
 * Created on 8 octobre 2002, 7:47
 */
/***
 * @author Vilvens
 */

public interface mobile
// Toute entité susceptible de se déplacer
{
    public void démarre ();
    public void avance (int d);
    public void tourne (int a);
    public void stoppe ();
}
```

Dès lors, on peut imaginer qu'un premier fabricant – les anglo-saxons 'computer oriented' disent "vendor" – crée une classe implémentant cet interface. Disons qu'il s'appelle "Goldorak" et que sa classe se nomme "tortue" (comme en Logo) :

tortue.java

```
/*
 * tortue.java
 */
/** 
 * @author Vilvens
 */

public class tortue implements mobile
{
    static public final String VENDOR = "Goldorak";
    boolean enMarche;
    boolean crayonPosé;
    int directionActuelle;

    // Utilitaire Debug ;)
    private void trace (String s)
    {
        System.out.println("T>>>> " + s);
    }

    // Constructeur par défaut - nécessaire pour newInstance()
    public tortue()
    {
        enMarche = false; crayonPosé = false; directionActuelle=0;
    }

    // Méthodes de l'interface
    public void démarre ()
    {
        enMarche = true;
        trace("Je suis prêt");
    }

    public void avance (int d)
    {
        if (!enMarche) return;
        trace("J'avance de " + d + " cm");
        if (crayonPosé) trace(" en laissant une trace");
        else trace(" sans laisser de trace");
    }

    public void tourne (int a)
    {
        if (!enMarche) return;
        trace("Je tourne de " + a + " °");
    }

    public void stoppe ()
    {
        enMarche = false;
        trace("Je suis arrêté");
    }
}
```

```
// Méthodes spécifiques
public void poseCrayon ()
{
    if (!enMarche) return;
    crayonPosé = true; trace("Je pose mon crayon sur le sol");
}
public void lèveCrayon ()
{
    if (!enMarche) return;
    crayonPosé = false; trace("Je relève mon crayon");
}
```

Mais un deuxième "vendor" a également créé une classe, plus modeste d'ailleurs, implémentant l'interface mobile. Disons que celui-là s'appelle "Charlotte" et que sa classe se nomme "routeurAPattes" (mouais ... bizarre, ces noms ...) :

routeurAPattes.java

```
/*
 * routeurAPattes.java
 */

/**
 * @author Vilvens
 */

public class routeurAPattes implements mobile
{
    static public final String VENDOR = "Charlotte";
    boolean ready;

    // Constructeur par défaut - nécessaire pour newInstance()
    public routeurAPattes()
    {
        ready = false;
    }

    // Utilitaire Debug
    private void trace (String s)
    {
        System.out.println("Rrrr>>>> " + s);
    }

    // Méthodes de l'interface
    public void démarre ()
    {
        ready = true;
        trace("Je suis prêt");
    }
```

```

public void avance (int d)
{
    if (!ready) return;
    trace("Je me déplace de " + d + " cm");
}
public void tourne (int a)
{
    if (!ready) return;
    trace("Je tourne de " + a + " °");
}
public void stoppe ()
{
    ready = false;
    trace("Je suis à l'arrêt");
}
}

```

Il paraît même qu'un troisième "vendor" nommé Genius-Vil travaille sur une classe fabuleuse du même genre – mais elle n'est pas encore disponible (mouais ...).

7.3 La méthode factory

Evidemment, le programme qui va décoder le script pourrait instancier à la dure l'une ou l'autre de ces classes. Mais il serait plus agréable de lui **transmettre, au début de son exécution, le type de robot à utiliser** et de le **laisser découvrir les éventuelles méthodes-fonctionnalités supplémentaires dont le robot est capable**. Nous allons donc considérer dans notre programme de traitement que le robot, appelons-le "tortueLogo", est un mobile, sans autre précision. Une méthode factory, style **getInstance()**, permettra dynamiquement, c'est-à-dire en cours d'exécution, de lui faire désigner un objet instanciant l'une ou l'autre classe.

L'interface mobile ne peut servir de siège à cette méthode factory, car celle-ci doit être manifestement une méthode de classe (static), pas d'instance. Nous allons donc créer une classe abstraite, disons **abstractMobile**, siège d'une méthode **getInstance()** dont le grand mérite sera de fournir un objet mobile instance de l'une des classes disponibles. Si ce n'est pas possible, la méthode factory lancera une exception :

abstractMobile.java

```

/*
 * abstractMobile.java
 * Created on 8 octobre 2002, 16:01
 */
/**
 * @author Vilvens
 */

public abstract class abstractMobile
{
    static mobile getInstance (String vendor) throws NoSuchMobileException
    {

```

```
if (vendor.equals("Goldorak"))
{
    tortue t = new tortue();
    return t;
}
else if (vendor.equals("Charlotte"))
{
    routeurAPattes r = new routeurAPattes();
    return r;
}
else throw new NoSuchMobileException("Vendor introuvable");
}
```

avec une classe d'exception des plus élémentaires :

NoSuchMobileException.java

```
/*
 * NoSuchMobileException.java
 * Created on 8 octobre 2002, 16:27
 */
/**
 * @author Vilvens (mouais ... pas de quoi s'en vanter ...)
 * @version
 */

public class NoSuchMobileException extends Exception
{
    public NoSuchMobileException (String m)
    {
        super(m);
    }
}
```

Bien sûr, on pourra objecter que le nom des vendors est écrit à la dure dans la classe abstraite. C'est vrai, et **il faudrait donc que la méthode factory aille lire dans un fichier le nom de la classe à instancier**, nom en regard du nom de son constructeur :

java.mobile

```
# Liste des vendeurs et de leur classe mobile
Goldorak : tortue
Charlotte : routeurAPattes
Sun : NonopetitRobot
```

ou, mieux encore, charger ce fichier dans un objet Properties pour y rechercher le vendeur. On pourrait aussi imaginer des classes **providers**. Absolument à faire ;-);-) ...

7.4 L'utilisation de l'API Reflection

Reconnaitre les différentes opérations-méthodes à exécuter sera un jeu d'enfant avec un programme de parsing à l'image de celui du paragraphe 4. Reste à modifier celui-ci pour invoquer les méthodes dont les noms sont reconnus.

La première chose à faire est d'obtenir de la factory un objet mobile. On suppose que le nom du vendor est lu sur la ligne de commande :

```
public static void main (String args[])
{
    if (args.length == 0)
    {
        trace("Veuillez entrer le nom du fabricant du robot sur la ligne de commande ...");
        System.exit(0);
    }

    vendor = args[0];
    ...
}
```

où vendor est une variable de classe de type String. Ainsi, lorsque l'on attaque le début du document XML (autrement dit dans la méthode startDocument()), on peut réclamer une instance de mobile par :

```
public void startDocument() throws SAXException
{
    try
    {
        tortueLogo = abstractMobile.getInstance(vendor);
    }
    catch (NoSuchMobileException e)
    {
        trace("Oh oh ... " + e.getMessage());
        System.exit(0);
    }
    ...
}
```

si tortueLogo est une variable membre déclarée comme étant un mobile :

```
public class XMLReflection extends DefaultHandler
{
    ...
    private mobile tortueLogo; //ce qui est commandé par le programme
    ...
}
```

Si l'on veut actionner les méthodes de l'objet ainsi obtenu, il faut à tout le moins savoir quelle classe il instancie (car mobile n'est qu'un interface). Cela peut se savoir au moyen de la méthode héritée d'Object :

```
public final Class getClass()
```

qui fournit un objet Class correspondant à la classe instanciée par un objet au moment de l'interprétation du bytecode. Ici :

```
public class XMLReflection extends DefaultHandler
{
    ...
    private Class info;
    ...

    public void startDocument() throws SAXException
    {
        try
        {
            tortueLogo = abstractMobile.getInstance(vendor);
        }
        ...
        info = tortueLogo.getClass();
        System.out.println("La classe véritable est " + info.getName());
        ...
    }
}
```

On constate que l'on peut obtenir le nom de la classe et d'ailleurs aussi le nom de ses méthodes :

```
private Method[] méthodes;
...
méthodes = info.getMethods();
System.out.println ("Liste des méthodes trouvées");
for (int i=0; i<méthodes.length; i++) System.out.println (méthodes[i].getName(););
...
```

Comme déjà dit, on voit ici que les méthodes de l'objet mystérieux peuvent être obtenues, si l'on peut dire, comme des objets au moyen de la méthode :

```
public Method[] getMethods() throws SecurityException
```

La classe **Method**, qui est définie dans le package `java.lang.reflect` et qui implémente l'interface `Member`, a évidemment pour rôle de fournir à la demande des informations sur la méthode considérée, comme par exemple son nom au moyen de la méthode :

```
public String getName()
```

ou le type de ses paramètres par :

```
public Class[] getParameterTypes()
```

7.5 L'exécution d'une instruction du script XML

Avec tous les éléments précédents, les choses deviennent claires. En effet, il suffira de :

- ♦ placer dans une liste (disons la variable membre *liste*, instance de **LinkedList**, classe de `java.util`) toutes les chaînes de caractères rencontrées dans le parsing du document XML – la méthode `characters()` est évidemment celle qui nous intéresse pour cela :

```
public void characters(char[] ch, int start,int length) throws SAXException
{
    String chaine = new String(ch, start, length).trim();
    if (chaine.length() > 0)
    {
        trace("@ Caractères", chaine);
        liste.addLast(chaine);
    }
}
```

- ♦ récupérer la méthode et les paramètres. Lorsque l'on parvient à la fin d'un élément XML "opération", nous savons donc que la liste contient en premier lieu le nom de l'instruction et ensuite, éventuellement, le(s) paramètre(s). Utiliser la méthode `invoke()`, évoquée ci-dessus, semble donc facile, mais

i) celle-ci réclame comme paramètre pour les arguments de la méthode un tableau d'Object; si

```
Vector paramètres = new Vector();
while (!liste.isEmpty()) paramètres.add(liste.removeFirst());
```

il nous faut donc utiliser la méthode

```
public Object[] toArray()
```

de la classe `Vector` pour obtenir le tableau attendu :

```
Object param[] = paramètres.toArray();
```

ii) certains paramètres de certaines méthodes sont des entiers (`int`), pas des chaînes de caractères – il nous faudra donc réaliser la conversion si nécessaire.

7.6 Le programme d'exécution du script robot

Epilogue : voici le programme de traitement complet ...

XMLReflection.java

```
/*
 * XMLReflection.java
 * Created on 1 octobre 2002, 15:48
 */
```

```

/***
 * @author Vilvens
 */

import javax.xml.parsers.*;
import org.xml.sax.*;
import org.xml.sax.helpers.*;

import java.io.*;
import java.util.*;

import java.lang.reflect.*;

public class XMLReflection extends DefaultHandler
{
    protected String nomFichierOpérations;
    static protected String vendor=null;

    private mobile tortueLogo; //ce qui est commandé par le programme

    private LinkedList liste;
    private Class info;
    private Method méthodes[];
    public String getNomFichierOpérations() { return nomFichierOpérations; }
    public void setNomFichierOpérations(String nfx) { nomFichierOpérations=nfx; }

    public XMLReflection(String nf)
    {
        setNomFichierOpérations(nf);
        tortueLogo=null; liste=null; info=null; méthodes=null;
    }

    // Utilitaires
    static protected void trace (String s)
    {
        System.out.println(s);
    }
    static protected void trace (String sCte, String s)
    {
        System.out.println(sCte + " : " + s);
    }
    static protected void trace (String sCte, int i)
    {
        System.out.println(sCte + " : " + i);
    }

    // Quelques méthodes du ContentHandler
    public void startDocument()throws SAXException
    {
        trace("## Début du script robot ##");
    }
}

```

```

try
{
    tortueLogo = abstractMobile.getInstance(vendor);
}
catch (NoSuchMobileException e)
{
    trace("Oh oh ... " + e.getMessage());
    System.exit(0);
}
trace("tortueLogo instanciée");

info = tortueLogo.getClass();
trace("La classe véritable est " + info.getName());

méthodes = info.getMethods();
trace("Liste des méthodes trouvées");
for (int i=0; i<méthodes.length; i++) trace(méthodes[i].getName());

liste = new LinkedList(); trace("Liste instanciée");
}

public void endDocument()throws SAXException
{
    trace("** Fin du script robot **");
}

public void endElement(java.lang.String uri, java.lang.String localName,
                     java.lang.String qName) throws SAXException
{
    if (qName.equals("opération") && !liste.isEmpty())
    {
        String nomMéthode = (String)liste.removeFirst();
        trace("Méthode invoquée", nomMéthode);

        Vector paramètres = new Vector();

        trace("Récupération des paramètres");
        if (liste.isEmpty() == true) trace("Liste des paramètres vide");
        while (!liste.isEmpty())
        {
            trace("Parcours de la liste des paramètres");
            paramètres.add(liste.removeFirst());
        }

        trace("Transformation des paramètres en tableau d'objets");
        Object param[] = paramètres.toArray();
        for (int i=0; i<param.length; i++)
            trace(i + ". " + param[i] + " / " + param[i].getClass().getName());
        if (param == null) trace ("Tableau des paramètres null");
        boolean méthodeTrouvée = false;
    }
}

```

```

trace("Parcours de la liste des méthodes");
for (int i=0; i<méthodes.length; i++)
{
    if (méthodes[i].getName().equals(nomMéthode))
    {
        try
        {
            méthodeTrouvée = true;
            trace("Méthode trouvée");
            Class typesParam[] = méthodes[i].getParameterTypes();
            for (int j=0; j<typesParam.length; j++)
            {
                String typeParam = typesParam[j].getName();
                trace("paramètre " + j + " : type = " + typeParam);
                if (typeParam.equals("int"))
                {
                    int x = Integer.parseInt(param[j].toString());
                    trace("Entier récupéré = " + x);
                    param[j]=new Integer(param[j].toString());
                }
            }
            Object ret = méthodes[i].invoke(tortueLogo, param);
            trace("Méthode invoquée");
            if (ret==null) trace("Pas de valeur de retour");
        }
        catch(NullPointerException e)
        {
            trace("Aie ! L'objet utilisé est une référence nulle", e.getMessage());
        }
        catch(IllegalAccessException e)
        {
            trace("Aie ! Tentative d'accès à une classe inaccessible", e.getMessage());
        }
        catch(IllegalArgumentException e)
        {
            trace("Aie ! Paramètres inadéquats pour la méthode", e.getMessage());
        }
        catch(InvocationTargetException e)
        {
            trace("Aie ! Problème dans l'exécution de la méthode", e.getMessage());
        }
        catch(ExceptionInInitializerError e)
        {
            trace("Aie ! Erreur dans un initialiseur statique", e.getMessage());
        }
    }
    if (!méthodeTrouvée)
        trace("????> Méthode " + nomMéthode + " non supportée !");
}
}

```

```

public void characters(char[] ch, int start,int length) throws SAXException
{
    String chaine = new String(ch, start, length).trim();
    if (chaine.length() > 0)
    {
        trace("@ Caractères", chaine);
        liste.addLast(chaine);
    }
}

public static void main (String args[])
{
    if (args.length == 0)
    {
        trace("Veuillez entrer lengthnom du fabricant du robot sur la ligne de commande ...");
        System.exit(0);
    }

    vendor = args[0];

    trace("Création du handler");
    XMLReflection sc = new XMLReflection
        ("D:\\java-forte-application\\XMLReflection\\listeOpérations.xml");
    trace("Création de la factory");
    SAXParserFactory spf = SAXParserFactory.newInstance();

    try
    {
        trace("Création du parser");
        SAXParser sp = spf.newSAXParser();
        trace("Mise en route du parsing \n\n");
        sp.parse(new File(sc.getNomFichierOpérations()), sc);
    }
    catch (ParserConfigurationException e)
    {
        System.out.println("Oh oh Problème de config : " + e.getMessage());
    }
    catch (SAXException e)
    {
        System.out.println("Oh oh Problème de SAX : " + e.getMessage());
    }
    catch (IOException e)
    {
        System.out.println("Oh oh Problème d'IO : " + e.getMessage());
    }
}
}

```

Un exemple d'exécution du programme avec "Goldorak" sur la ligne de commande donne :

Création du handler
Création de la factory
Création du parser
Mise en route du parsing
**** Début du script robot ****
tortueLogo instanciée
La classe véritable est **tortue**
Liste des méthodes trouvées
hashCode
wait
wait
wait
getClass
equals
toString
notify
notifyAll
stoppe
tourne
avance
démarre
poseCrayon
lèveCrayon
Liste instanciée
@ Caractères : démarre
Méthode invoquée : démarre
Récupération des paramètres
Liste des paramètres vide
Transformation des paramètres en tableau d'objets
Parcours de la liste des méthodes
Méthode trouvée
T>>>> Je suis prêt
Méthode invoquée
Pas de valeur de retour
@ Caractères : poseCrayon
Méthode invoquée : poseCrayon
Récupération des paramètres
Liste des paramètres vide
Transformation des paramètres en tableau d'objets
Parcours de la liste des méthodes
Méthode trouvée
T>>>> Je pose mon crayon sur le sol
Méthode invoquée
Pas de valeur de retour
@ Caractères : tourne
@ Caractères : 30
Méthode invoquée : tourne
Récupération des paramètres
Parcours de la liste des paramètres
Transformation des paramètres en tableau d'objets

```
0. 30 / java.lang.String
Parcours de la liste des méthodes
Méthode trouvée
paramètre 0 : type = int
Entier récupéré = 30
T>>>> Je tourne de 30 °
Méthode invoquée
Pas de valeur de retour
@ Caractères : avance
@ Caractères : 50
Méthode invoquée : avance
Récupération des paramètres
Parcours de la liste des paramètres
Transformation des paramètres en tableau d'objets
0. 50 / java.lang.String
Parcours de la liste des méthodes
Méthode trouvée
paramètre 0 : type = int
Entier récupéré = 50
T>>>> J'avance de 50 cm
T>>>> en laissant une trace
Méthode invoquée
Pas de valeur de retour
@ Caractères : tourne
@ Caractères : 60
Méthode invoquée : tourne
Récupération des paramètres
Parcours de la liste des paramètres
Transformation des paramètres en tableau d'objets
0. 60 / java.lang.String
Parcours de la liste des méthodes
Méthode trouvée
paramètre 0 : type = int
Entier récupéré = 60
T>>>> Je tourne de 60 °
Méthode invoquée
Pas de valeur de retour
@ Caractères : avance
@ Caractères : 80
Méthode invoquée : avance
Récupération des paramètres
Parcours de la liste des paramètres
Transformation des paramètres en tableau d'objets
0. 80 / java.lang.String
Parcours de la liste des méthodes
Méthode trouvée
paramètre 0 : type = int
Entier récupéré = 80
T>>>> J'avance de 80 cm
T>>>> en laissant une trace
```

Méthode invoquée
Pas de valeur de retour
@ Caractères : lèveCrayon
Méthode invoquée : lèveCrayon
Récupération des paramètres
Transformation des paramètres en tableau d'objets
Parcours de la liste des méthodes
Méthode trouvée
T>>>> Je relève mon crayon
Méthode invoquée
Pas de valeur de retour
@ Caractères : monte
Méthode invoquée : monte
Récupération des paramètres
Transformation des paramètres en tableau d'objets
Parcours de la liste des méthodes
????> Méthode monte non supportée !
@ Caractères : stoppe
Méthode invoquée : stoppe
Récupération des paramètres
Liste des paramètres vide
Transformation des paramètres en tableau d'objets
Parcours de la liste des méthodes
Méthode trouvée
T>>>> Je suis arrêté
Méthode invoquée
Pas de valeur de retour
**** Fin du script robot ****

On remarquera qu'une méthode demandée par le script XML ne provoque pas de fin prématuée du programme mais seulement un message d'erreur. Si nous utilisons maintenant le vendeur "Charlotte", cela donne :

Création du handler
Création de la factory
Création du parser
Mise en route du parsing

**** Début du script robot ****
tortueLogo instanciée
La classe véritable est **routeurAPattes**
Liste des méthodes trouvées
hashCode
wait
wait
wait
getClass
equals
toString

notify
notifyAll
stoppe
tourne
avance
démarre
Liste instanciée
@ Caractères : démarre
Méthode invoquée : démarre
Récupération des paramètres
Liste des paramètres vide
Transformation des paramètres en tableau d'objets
Parcours de la liste des méthodes
Méthode trouvée
Rrrr>>>> Je suis prêt
Méthode invoquée
Pas de valeur de retour
@ Caractères : poseCrayon
Méthode invoquée : poseCrayon
Récupération des paramètres
Liste des paramètres vide
Transformation des paramètres en tableau d'objets
Parcours de la liste des méthodes
????> Méthode poseCrayon non supportée !
@ Caractères : tourne
@ Caractères : 30
Méthode invoquée : tourne
Récupération des paramètres
Parcours de la liste des paramètres
Transformation des paramètres en tableau d'objets
0. 30 / java.lang.String
Parcours de la liste des méthodes
Méthode trouvée
paramètre 0 : type = int
Entier récupéré = 30
Rrrr>>>> Je tourne de 30 °
Méthode invoquée
Pas de valeur de retour
@ Caractères : avance
@ Caractères : 50
Méthode invoquée : avance
Récupération des paramètres
Parcours de la liste des paramètres
Transformation des paramètres en tableau d'objets
0. 50 / java.lang.String
Parcours de la liste des méthodes
Méthode trouvée
paramètre 0 : type = int
Entier récupéré = 50
Rrrr>>>> Je me déplace de 50 cm

Méthode invoquée
Pas de valeur de retour
@ Caractères : tourne
@ Caractères : 60
Méthode invoquée : tourne
Récupération des paramètres
Parcours de la liste des paramètres
Transformation des paramètres en tableau d'objets
0. 60 / java.lang.String
Parcours de la liste des méthodes
Méthode trouvée
paramètre 0 : type = int
Entier récupéré = 60
Rrrr>>>> Je tourne de 60 °
Méthode invoquée
Pas de valeur de retour
@ Caractères : avance
@ Caractères : 80
Méthode invoquée : avance
Récupération des paramètres
Parcours de la liste des paramètres
Transformation des paramètres en tableau d'objets
0. 80 / java.lang.String
Parcours de la liste des méthodes
Méthode trouvée
paramètre 0 : type = int
Entier récupéré = 80
Rrrr>>>> Je me déplace de 80 cm
Méthode invoquée
Pas de valeur de retour
@ Caractères : lèveCrayon
Méthode invoquée : lèveCrayon
Récupération des paramètres
Liste des paramètres vide
Transformation des paramètres en tableau d'objets
Parcours de la liste des méthodes
????> Méthode lèveCrayon non supportée !
@ Caractères : monte
Méthode invoquée : monte
Récupération des paramètres
Liste des paramètres vide
Transformation des paramètres en tableau d'objets
Parcours de la liste des méthodes
????> Méthode monte non supportée !
@ Caractères : stoppe
Méthode invoquée : stoppe
Récupération des paramètres
Liste des paramètres vide
Transformation des paramètres en tableau d'objets
Parcours de la liste des méthodes

Méthode trouvée

Rrrr>>>> Je suis à l'arrêt

Méthode invoquée

Pas de valeur de retour

** Fin du script robot **

On remarquera que les méthodes `lèveCrayon()` et `poseCrayon()`, fournie par tortue mais ne faisant pas partie de l'interface mobile, peuvent être demandées sans autre risque que le constat de leur absence.

Enfin, si l'on fait appel au formidable Genius-Vil, cela donne :

Création du handler

Création de la factory

Création du parser

Mise en route du parsing

**** Début du script robot ****

Oh oh ... Vendor introuvable

Logique ...

8. Un exemple de parser DOM

8.1 Les documents DOM

DOM utilise la structure manifestement hiérarchique d'un document XML pour en créer en mémoire une représentation arborescente sur laquelle toutes les opérations (navigation, insertion, remplacement, ...) seront possibles. Tous les éléments d'un fichier XML deviennent ainsi les noeuds d'un arbre qui correspond à un objet implémentant l'interface **Document**. Le noyau de base de DOM (le "core") définit dans le package `org.w3c.dom`, outre l'interface Document, un ensemble d'interfaces implémentés par les objets noeuds. Les noeuds peuvent en fait être de différents types :

1) les noeuds de structure (read-only)

Il s'agit de noeuds qui ne constituent pas des données textuelles ou des contenus d'éléments, mais qui correspondent cependant à des structures syntaxiques bien présentes dans le document XML. Les types de cette catégorie sont :

- ◆ DOCUMENT_TYPE : correspondant à la déclaration `<!DOCTYPE>`, un tel noeud permet d'accéder aux notations, aux entités, aux identifiants public et système de la définition du type de document;
- ◆ PROCESSING_INSTRUCTION : correspondant à des instructions de traitement au sein du document XML;
- ◆ NOTATION : pour les entités externes non parsées;
- ◆ ENTITY : nom d'une entité de référence.

2) **les nœuds de contenu** (read-write)

Les nœuds de ce type sont ceux qui correspondent aux données effectivement contenues dans le fichier XML. Les types existant sont :

- ◆ DOCUMENT : ce type de nœud est unique pour un document XML, puisqu'il procure l'accès au type du document et à son nœud de départ qui contient en fait le corps complet du document parse.
- ◆ ELEMENT : ce type de nœud est sans doute le plus fréquent, puisque de tels nœuds vont permettre l'accès à l'information effective : ils sont parents des nœuds des 4 types suivants :
 - TEXT : données textuelles
 - COMMENT : commentaires XML
 - ENTITY_REFERENCE : entités de référence (du genre """)
 - CDATA_SECTION : pour des caractères qui sont normalement des caractères intervenant dans le balisage.
- ◆ DOCUMENT_FRAGMENT_NODE : pour matérialiser un morceau de document XML qui aurait été conservé par un couper-coller.
- ◆ ATTRIBUTE : il s'agit évidemment des attributs des tags XML; ils possèdent un nœud enfant TEXT contenant la valeur textuelle en question.

Concrètement, voyons à présent comment parser un document XML en utilisant DOM.

8.2 **Obtenir un parser DOM**

A nouveau, pour instancier le parser lui-même, il nous faudra en obtenir une instance au moyen d'une factory. La *classe factory* se nomme **DocumentBuilderFactory** (du package javax.xml.parsers) et comporte toutes les méthodes permettant de créer et de configurer un parser qui produira des arbres DOM. Parmi ces méthodes, relevons par exemple les commandes de configuration :

```
public void setValidating(boolean validating)
```

Pour spécifier que le parser est aussi un validateur (par défaut, ce n'est pas le cas).

```
public void setNamespaceAware(boolean awareness)
```

Pour spécifier que le parser sait gérer les namespaces (par défaut, ce n'est pas le cas). Mais ce qui nous intéresse le plus directement est bien sûr la méthode permettant d'obtenir la factory :

```
public static DocumentBuilderFactory newInstance()
```

qui permet d'obtenir une instance de **DocumentBuilderFactory**, indépendamment de la manière dont cette instance est effectivement implémentée. En fait, cette méthode de classe va successivement rechercher la classe implementant la factory

- ◆ dans la propriété du système nommée javax.xml.parsers. DocumentBuilderFactory;
- ◆ dans le fichier lib/jaxp.properties du JRE;

- ◆ dans les API de service (Services API) qui sont éventuellement définies dans META-INF/services/javax.xml.parsers.DocumentBuilderFactory (c'est le cas dans crimson.jar);
- ◆ dans l'instance de DocumentBuilderFactory par défaut pour le système.

Donc, finalement :

```
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
```

2) Cette classe factory va maintenant nous permettre d'obtenir une instance du parser attendu, instance de la classe **DocumentBuilder**, en utilisant la méthode

```
public abstract DocumentBuilder newDocumentBuilder()  
throws ParserConfigurationException
```

qui utilise les valeurs des paramètres définies par les méthodes de SAXParserFactory (par exemple, setValidating(true)). Le parsing en lui-même est demandé au moyen de la méthode

```
public Document parse(java.io.File f) throws SAXException, java.io.IOException
```

qui possède une armée de méthodes sœurs polymorphes. Cette méthode va donc produire un objet qui implémente l'interface **Document** (lui même dérivé de l'interface **Node** au sein du package org.w3c.dom) qui représente l'arbre DOM en mémoire. On remarquera que la méthode parse() est susceptible de renvoyer l'exception SAXException : en fait, bon nombre de méthodes de la classe DocumentBuilder sont basées sur des méthodes de SAX.

Donc, finalement, si doc est un objet Document :

```
DocumentBuilder db = dbf.newDocumentBuilder();  
doc = db.parse(new File("D:\\java-forte-application\\XmlSaxParser\\listeSociétés.xml"));
```

8.3 Obtenir de l'information sur les nœuds

Afin d'introspecter l'arbre DOM qui a été créé en mémoire (rien de visible sur la console ne nous confirme ce fait), nous allons utiliser diverses méthodes de Document, ou plutôt de l'interface parent Node :

- ◆ public short getNodeType()

Retourne un code correspondant au type de nœud – les types dont il s'agit sont ceux utilisé par DOM, comme par exemple ELEMENT ou ATTRIBUTE. Tous ces codes sont encapsulés dans l'interface Node au moyen de constantes de classe :

```
public static final short ELEMENT_NODE  
public static final short ATTRIBUTE_NODE  
public static final short TEXT_NODE  
public static final short CDATA_SECTION_NODE  
public static final short ENTITY_REFERENCE_NODE  
public static final short ENTITY_NODE  
public static final short PROCESSING_INSTRUCTION_NODE  
public static final short COMMENT_NODE
```

```
public static final short DOCUMENT_NODE  
public static final short DOCUMENT_TYPE_NODE  
public static final short DOCUMENT_FRAGMENT_NODE  
public static final short NOTATION_NODE
```

- ◆ **public java.lang.String getNodeName()**

Retourne le nom du nœud.

- ◆ **public java.lang.String getNodeValue() throws DOMException**

Retourne la valeur du nœud qui s'interprète en fonction du type du nœud. On remarquera l'exception susceptible d'être lancée : la classe **DOMException** (du package `org.w3c.dom`) est dérivée non pas, comme le plus souvent, d'`Exception`, mais de `RuntimeException` (du package `java.lang`). La classe contient toute une série de constantes de classe qu'il faut bien appeler des codes d'erreur (-) ..), comme par exemple :

```
public static final short DOMSTRING_SIZE_ERR
```

lorsqu'une chaîne est trop longue pour être mémorisée dans une chaîne du modèle DOM
public static final short NO_MODIFICATION_ALLOWED_ERR
lorsqu'un nœud est read-only

- ◆ **public NodeList getChildNodes()**

Fournit la liste des noeuds enfants du nœud considéré; la liste implémente l'interface **NodeList** (toujours du même package `org.w3c.dom`) qui est une collection ordonnée. On accède aux éléments par un indice, dont la valeur minimale est 0 et dont la valeur maximale est connue par

`public int getLength(),`

au moyen de la méthode :

`public Node item(int index)`

- ◆ **public NamedNodeMap getAttributes()**

Récupère les attributs du nœud dans un objet qui implémente l'interface **NamedNodeMap** (encore du même package `org.w3c.dom`). Un tel objet est une simple collection non ordonnée : clairement, elle constitue un simple réceptacle qui contiendra les attributs rencontrés, sans ordre en rapport avec une sémantique quelconque. Tout comme pour la **NodeList**, on accède aux éléments par un indice, dont la valeur maximale est connue par

`public int getLength(),`

au moyen de la méthode :

`public Node item(int index)`

8.4 La visualisation du document DOM

Avec de telles armes, et sans composant graphique, il devient possible de visualiser l'arbre DOM sur la console. En effet, il suffit de créer une méthode, disons `afficheArbreEnTexte()`, qui parcourt récursivement cet arbre : pour chaque nœud rencontré,

- ◆ on extrait le nom;
- ◆ on extrait la valeur;
- ◆ on rappelle la méthode pour chaque enfant du nœud courant.

Pour que l'affichage soit plus parlant, on peut afficher chaque nœud avec un retrait proportionnel à la profondeur de sa position dans l'arbre. Il suffit de passer à `afficheArbreEnTexte()` un second paramètre qui représente cette profondeur; elle sera donc incrémentée de 1 à chaque nouvel appel. Une simplissime méthode d'affichage `println()` tiendra compte de cette profondeur pour générer le retrait correspondant. Donc :

XmlDomParser.java

```
/*
 * XmlDomParser.java
 * Created on 20 août 2002, 15:54
 */

package parsers;

/**
 * @author Vilvens
 * @version
 */

import javax.xml.parsers.*;
import org.w3c.dom.*;
import org.xml.sax.*;

import java.io.*;

public class XmlDomParser
{
    protected String nomFichierXML;
    /** Creates new XmlDomParser */
    public XmlDomParser (String nf)
    {
        nomFichierXML = nf;
    }

    public static void afficheArbreEnTexte (Node noeud, int profondeur)
    {
        if (noeud == null) return; // arrêt de la descente récursive

        String nom = noeud.getnodeName();
        String valeur = noeud.getNodeValue();
```

```

//toiletage
if (valeur != null) valeur = valeur.trim();
// éliminations des textes vides
if (noeud.getNodeType() == Node.TEXT_NODE && valeur.equals("") ) ;
else println(nom + " : " + valeur, profondeur);

NodeList enfants = noeud.getChildNodes();
for (int i=0; i<enfants.getLength(); i++)
{
    // appel récursif pour chaque enfant
    afficheArbreEnTexte (enfants.item(i), profondeur+1);
}
}

public static void println (String s, int retrait)
{
    retrait *=2; // pour un décalage plus marqué
    for (int i=0; i<retrait; i++) System.out.print(" ");
    System.out.println(s);
}

public static void main (String args[])
{
    Document doc;

    DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
    dbf.setValidating(true);
    dbf.setNamespaceAware(true);

    try
    {
        DocumentBuilder db = dbf.newDocumentBuilder();
        doc = db;
        parse(new File("c:\\java-netbeans-application\\XMLParsers\\src\\listeSociétés.xml"));
        afficheArbreEnTexte (doc, 0);
    }
    catch (ParserConfigurationException e)
    {
        System.out.println("Oh oh Problème de configuration du parser ..." + e.getMessage());
    }
    catch (SAXException e)
    {
        System.out.println("Oh oh Problème de SAX : " + e.getMessage());
    }
    catch (IOException e)
    {
        System.out.println("Oh oh Problème d'IO : " + e.getMessage());
    }
}
}

```

L'application de notre programme au fichier XML de sociétés donne :

```
#document : null
#comment : Created by Vilvens on 19 août 2002, 12:16
listeSociétés : null
listeSociétés : null
société : null
nomSociété : null
#text : Genius S.A.
adresse : null
rue : null
#text : rue de la Réussite
numéro : null
#text : 113
codePostal : null
#text : 4500
ville : null
#text : Visé
contact : null
nom : null
#text : Bidendum
prénom : null
#text : Albert
fonction : null
#text : General Manager
societe : null
nomSociété : null
#text : Mes belles mîches S.P.R.L.
adresse : null
rue : null
#text : rue de la Boulangerie
numéro : null
#text : 69
...
numéroEnregistrement : null
#text : 502/4781/80001
```

8.5 Le type des noeuds

On se souviendra encore qu'il est possible d'obtenir le type des nœuds parcourus. Cependant, ces types sont fournis par l'intermédiaire d'un code numérique. Nous allons donc nous doter d'une variable membre tableau statique qui permettra de transformer un code en la chaîne de caractères correspondante plus compréhensible :

```
private static String typesNoeuds[] =
{ "", "ELEMENT", "ATTRIBUTE", "TEXT", "CDATA_SECTION",
"ENTITY_REFERENCE", "ENTITY", "PROCESSING_INSTRUCTION", "COMMENT",
"DOCUMENT", "DOCUMENT_TYPE", "DOCUMENT_FRAGMENT", "NOTATION"};
```

Il suffit alors de demander également l'affichage du type par :

```
public static void afficheArbreEnTexte (Node noeud, int profondeur)
{
    if (noeud == null) return; // arrêt de la descente récursive
    ...
    if (noeud.getNodeType() == Node.TEXT_NODE && valeur.equals("") ) ;
    else println(nom + " : " + (valeur!=null?valeur:"") + "(" +
                 typesNoeuds[noeud.getNodeType()] + ")");
}
```

Le résultat est alors :

```
#document : null (DOCUMENT)
#comment : Created by Vilvens on 19 août 2002, 12:16 (COMMENT)
listeSociétés : null (DOCUMENT_TYPE)
listeSociétés : null (ELEMENT)
société : null (ELEMENT)
nomSociété : null (ELEMENT)
#text : Genius S.A. (TEXT)
adresse : null (ELEMENT)
rue : null (ELEMENT)
#text : rue de la Réussite (TEXT)
numéro : null (ELEMENT)
#text : 113 (TEXT)
codePostal : null (ELEMENT)
#text : 4500 (TEXT)
ville : null (ELEMENT)
#text : Visé (TEXT)
contact : null (ELEMENT)
nom : null (ELEMENT)
#text : Bidendum (TEXT)
prénom : null (ELEMENT)
#text : Albert (TEXT)
fonction : null (ELEMENT)
#text : General Manager (TEXT)
...
```

Notre méthode `afficheArbreEnTexte()` devra normalement encore être complétée par l'analyse des attributs au moyen de la méthode `getAttributes()`.

9. La création directe d'un arbre DOM

9.1 La création en mémoire

Dans la présentation générale, nous avons dit que les APIs DOM permettaient non seulement de parser un document XML existant, mais aussi de le modifier ou même de créer un nouveau document DOM à partir de rien. Voyons cela d'un peu plus près.

Imaginons donc que nous souhaitions disposer d'une application au look suivant :



L'objectif est bien clair ...

Comme dans le programme précédent, nous allons nous tout d'abord fournir un objet DocumentBuilder, dont nous utiliserons la méthode :

```
public abstract Document newDocument()
```

qui nous fournit un objet implémentant Document. Donc :

```
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
dbf.setValidating(false);
try
{
    DocumentBuilder db = dbf.newDocumentBuilder();
    documentCree = db.newDocument();
}
...
```

Bien sûr, il est vide. Nous allons lui donner un élément de base au moyen des méthodes :

```
public Element createElement(java.lang.String tagName) throws DOMException
```

Permet de créer un tag – si le nom utilisé contient des caractères illégaux pour un nom de tag (un blanc ou une apostrophe par exemple), l'exception DOMException est lancée. Le tag est un objet qui implémente l'interface **Element** dérivé de Node.

```
public Node appendChild(Node newChild) throws DOMException
```

Héritée de l'interface Node, cette méthode permet au nœud qui l'appelle de se donner un nouveau noeud enfant.

Donc :

```
Element racine = documentCree.createElement("Societe");  
documentCree.appendChild(racine);
```

Nous allons de plus gérer une table de hachage (appelons-la "lesNoeuds") dans le but de trouver facilement par son nom le nœud d'un chef immédiat; accrocher le nouveau nœud sera alors un jeu d'enfant. Au départ, seul l'élément racine y figure :

```
lesNoeuds.put("Societe", racine);
```

Par la suite, nous rechercherons le nœud du nom du chef dans la table de hachage et, en cas de succès, nous accrocherons le nouveau noeud créé au noeud proprement dit du chef. Celui-ci est le père du nœud contenant comme information textuelle le nom cherché :

```
Node pere = (Node) lesNoeuds.get(chef);  
pere.getParentNode().appendChild(el);
```

Il ne faudra pas oublier d'ajouter à ce noeud un nœud enfant contenant l'information textuelle associée, ceci au moyen de la méthode :

```
public Text createTextNode(java.lang.String data)
```

qui renvoie un objet implémentant l'interface **Text**; son seul mérite est de représenter, en sémantique XML, une information textuelle. Donc :

```
elTexte = documentCree.createTextNode(nom);  
el.appendChild(elTexte);  
lesNoeuds.put(nom, elTexte);
```

Sans trop insister, cela donne :

XmldomTreeCreate.java

```
/*  
 * XmldomTreeCreate.java  
 * Created on 22 août 2007, 8:48  
 */
```

```
package parsers;
```

```
import java.awt.*;  
import java.util.*;
```

```

import org.w3c.dom.*;
import javax.xml.parsers.*;

/**
 * @author Vilvens
 */

public class XmlDomTreeCreate extends javax.swing.JFrame
{
    static Document documentCree = null;
    static Hashtable lesNoeuds = new Hashtable();
    private static String typesNoeuds[] =
    {"ELEMENT", "ATTRIBUTE", "TEXT", "CDATA_SECTION",
     "ENTITY_REFERENCE", "ENTITY", "PROCESSING_INSTRUCTION",
     "COMMENT", "DOCUMENT", "DOCUMENT_TYPE",
     "DOCUMENT_FRAGMENT", "NOTATION"};

    public XmlDomTreeCreate() {initComponents(); }
    private void initComponents() { ... }

    private void BFiniActionPerformed(java.awt.event.ActionEvent evt)
    {
        System.exit(0);
    }

    private void BVisualiserActionPerformed(java.awt.event.ActionEvent evt)
    {
        System.out.println("Visualisation de l'arbre");
        afficheArbreEnTexte(documentCree, 0);
    }

    private void BAjouterActionPerformed(java.awt.event.ActionEvent evt)
    {
        System.out.println("Ajout d'un élément");

        String nom = TFNouveauNom.getText();
        String fonction = TFFonction.getText();
        String chef = TFChefImmediat.getText();
        System.out.println("fonction lue = "+fonction);
        Element el= documentCree.createElement(fonction);
        System.out.println("Elément créé");

        String attribut = TFAttribut.getText();
        if (!attribut.equals(""))
        {
            String valAttribut = TFAValeurAttribut.getText();
            el.setAttribute(attribut, valAttribut);
        }
        Text elTexte = null;
    }
}

```

```

try
{
    if (lesNoeuds.containsKey(chef))
    {
        System.out.println("Chef trouvé");
        Node pere = (Node) lesNoeuds.get(chef);
        pere.getparentNode().appendChild(el);
    }
    else
    {
        System.out.println("Chef non trouvé");
        documentCree.getLastChild().appendChild(el);
    }

    elTexte = documentCree.createTextNode(nom);
    el.appendChild(elTexte);
    System.out.println("Elément ajouté");
    lesNoeuds.put(nom, elTexte);
}
catch (DOMException e)
{
    System.out.println("oh oh ? " + e.getMessage() + "\nElément non ajouté");
}
afficheArbreEnTexte(documentCree, 0);
}

public static void afficheArbreEnTexte (Node noeud, int profondeur)
{
    if (noeud == null) return;
    String nom = noeud.getNodeName();
    String valeur = noeud.getNodeValue();
    if (valeur != null) valeur = valeur.trim();
    if (noeud.getNodeType() == Node.TEXT_NODE && valeur.equals("") );
    else println(nom + " : " + (valeur!=null?valeur:"") + " (" +
        typesNoeuds[noeud.getNodeType()] + ")");
    NodeList enfants = noeud.getChildNodes();
    for (int i=0; i<enfants.getLength(); i++)
    {
        afficheArbreEnTexte (enfants.item(i), profondeur+1);
    }
}

public static void println (String s, int retrait)
{
    retrait *=2;
    for (int i=0; i<retrait; i++) System.out.print(" ");
    System.out.println(s);
}

```

```
public static void main(String args[])
{
    java.awt.EventQueue.invokeLater(new Runnable()
    {
        public void run()
        {
            new XmlDomTreeCreate().setVisible(true);

            DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
            dbf.setValidating(false);
            try
            {
                DocumentBuilder db = dbf.newDocumentBuilder();
                documentCree = db.newDocument();
            }
            catch (ParserConfigurationException e)
            {
                System.out.println("Oh oh Problème de configuration du parser ..." +
                    e.getMessage());
            }
            Element racine = documentCree.createElement("Societe");
            documentCree.appendChild(racine);
            afficheArbreEnTexte(documentCree, 0);
            lesNoeuds.put("Societe", racine);
        }
    });
}

private javax.swing.JButton BAjouter;
private javax.swing.JButton BFini;
private javax.swing.JButton BVisualiser;
private javax.swing.JTextField TFAtribut;
private javax.swing.JTextField TFChefImmediat;
private javax.swing.JTextField TFFonction;
private javax.swing.JTextField TFNouveauNom;
private javax.swing.JTextField TFValeurAttribut;
...
private javax.swing.JLabel jLabel6;
}
```

Un exemple d'exécution pourrait être une suite successive d'insertions :

Ajout d'un employé :

Nom :	vilvens
Fonction :	guide
Chef immédiat :	aucun
Attribut :	
Valeur attribut :	

Ajout d'un employé :

Nom :	myriam
Fonction :	épouse
Chef immédiat :	vilvens
Attribut :	
Valeur attribut :	

donne sur la console des résultats du type :

```

Usages                               Output
XML check × XMLParsers (run-single) ×

Ajout d'un élément
fonction lue = guide
Elément créé
Chef non trouvé
Elément ajouté
#document : (DOCUMENT)
  Societe : (ELEMENT)
    guide : (ELEMENT)
      #text : vilvens (TEXT)

Usages                               Output
XML check × XMLParsers (run-single) ×
Ajout d'un élément
fonction lue = épouse
Elément créé
Chef trouvé
Elément ajouté
#document : (DOCUMENT)
  Societe : (ELEMENT)
    guide : (ELEMENT)
      #text : vilvens (TEXT)
    épouse : (ELEMENT)
      #text : myriam (TEXT)

```

avec un affichage pour la visualisation finale de l'arbre :

```

Usages          Output
XML check      XMLParsers (run-single)
Visualisation de l'arbre
#document : (DOCUMENT)
    Societe : (ELEMENT)
        guide : (ELEMENT)
            #text : vilvens (TEXT)
            épouse : (ELEMENT)
                #text : myriam (TEXT)
            fils : (ELEMENT)
                #text : denys (TEXT)
            fils : (ELEMENT)
                #text : pierre (TEXT)
        copine : (ELEMENT)
            #text : francesca (TEXT)
        collègue : (ELEMENT)
            #text : albert (TEXT)

```

Evidemment, c'est du mode texte – bof ... un JTree eût donné un résultat bien plus beau (to do ;-)).

9.2 La sauvegarde dans un fichier XML (1)

Notre arbre DOM se trouve toujours en mémoire. Le plus souvent, nous souhaiterons le sauvegarder dans un fichier XML : on peut encore parler de "sérialisation". Il existe pour ce faire un utilitaire de sérialisation dont les classes se trouvent dans le package org.apache.xml.serialize (fourni dans xerces.jar). La classe de base est la classe **XMLSerializer** qui, par sa classe mère abstraite **BaseMarkupSerializer**, implémente en fait l'interface **Serializer**. Le grand mérite de cette classe, on s'en doute vu son nom, est de fournir une méthode

```
public void serialize (Document doc) throws java.io.IOException
```

dont le rôle est évidemment de sauvegarder le document DOM dans un fichier. Celui-ci, personnalisé évidemment par un flux, est précisé comme premier paramètre dans l'un des constructeurs de la classe :

```
public XMLSerializer (OutputStream output, OutputFormat format)
```

Le deuxième paramètre, instance de la classe **OutputFormat**, permet de paramétriser la sérialisation quant à la plate-forme d'encodage (pour nous, classiquement, ISO-8859-1) et au fait d'insérer des indentations ou pas – l'un des constructeurs enregistre ces paramètres :

```
public OutputFormat (Document doc, String encoding, boolean indenting)
```

Nous pouvons dès lors ajouter à notre programme précédent un bouton supplémentaire permettant de sauvegarder. Le code additionnel est alors :

```
import java.io.*;
import org.apache.xml.serialize.*;

private void BSauverSerializeActionPerformed (java.awt.event.ActionEvent evt)
{
    System.out.println("Ecriture de l'arbre DOM dans un fichier DomTree.xml");
    try
    {
        OutputFormat of = new OutputFormat (documentCree, "ISO-8859-1", true);
        FileOutputStream fos = new FileOutputStream (
            "c:\\java-forte-application\\XmlDomTreeCreate\\DomTree.xml");
        BaseMarkupSerializer bms = new XMLSerializer (fos, of);
        bms.serialize(documentCree);
    }
    catch (Exception e)
    {
        System.out.println("Erreur de sérialisation : " + e.getMessage());
    }
}
```

Une exécution analogue à la précédente donne sur la console :

Visualisation de l'arbre
#document : (DOCUMENT)
Societe : (ELEMENT)
guide : (ELEMENT)
#text : vilvens (TEXT)
épouse : (ELEMENT)
formes : sculpturales (ATTRIBUTE)
#text : myriam (TEXT)
fils : (ELEMENT)
comportement : turbulent (ATTRIBUTE)
#text : denys (TEXT)
fils : (ELEMENT)
comportement : calme (ATTRIBUTE)
#text : pierre (TEXT)
copine : (ELEMENT)
formes : sculpturales aussi (ATTRIBUTE)
#text : francesca (TEXT)
collègue : (ELEMENT)
#text : albert (TEXT)

tandis que le fichier résultant est bien ce que l'on attend :

DomTree.xml
<?xml version="1.0" encoding="ISO-8859-1"?> <Societe> <guide>vilvens <épouse formes="sculpturales">myriam

```
<fils comportement="turbulent">denys</fils>
<fils comportement="calme">pierre</fils>
</épouse>
<copine formes="sculpturales aussi">francesca</copine>
<collègue>albert</collègue>
</guide>
</Societe>
```

En fait, il est aussi possible d'obtenir ce résultat avec un Transformer : qu'est-ce donc ?

10. Les documents XSL sous NetBeans

On se souviendra que les fichiers XML peuvent faire l'objet de traitements que l'on appelle des transformations : **XSLT** résume la norme correspondante. Considérons par exemple un fichier XML (listSoc.xml), du type du fichier XML des sociétés évoqué plus haut :

listSoc.xml

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<!-- Created by Vilvens on 30 août 2002, 11:39 --&gt;

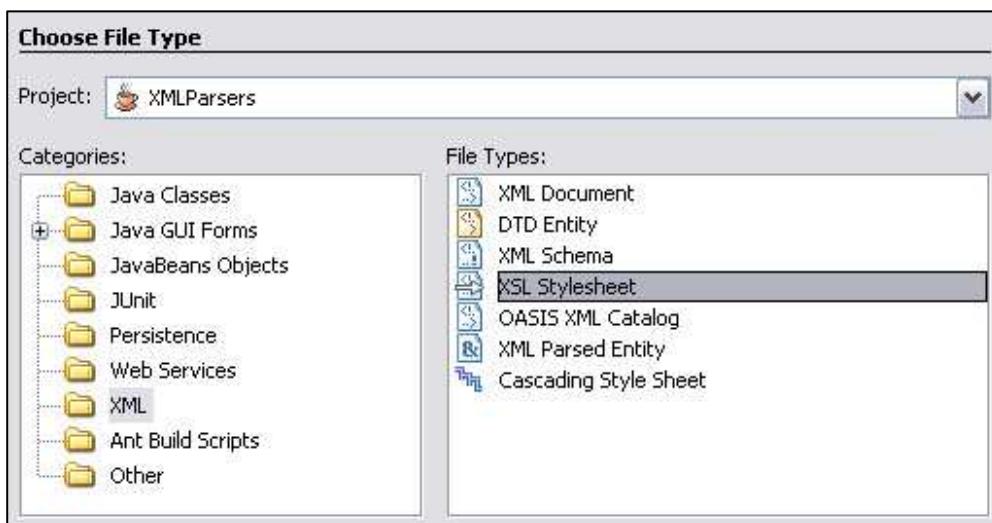
&lt;listSoc&gt;
    &lt;société&gt;
        &lt;nomSociété&gt;Genius S.A.&lt;/nomSociété&gt;
        &lt;adresse&gt;
            &lt;rue&gt;rue de la Réussite&lt;/rue&gt;
            &lt;numéro&gt;113&lt;/numéro&gt;
            &lt;codePostal&gt;4500&lt;/codePostal&gt;
            &lt;ville&gt;Visé&lt;/ville&gt;
        &lt;/adresse&gt;
        &lt;contact&gt;
            &lt;nom&gt;Bidendum&lt;/nom&gt;
            &lt;prénom&gt;Albert&lt;/prénom&gt;
            &lt;fonction&gt;General Manager&lt;/fonction&gt;
            &lt;email&gt;abidend@groumf.com&lt;/email&gt;
            &lt;email&gt;albert@sexy.net&lt;/email&gt;
        &lt;/contact&gt;
        &lt;numéroEnregistrement ONSS="AG007" interne="BOND"&gt;500/213/40001
        &lt;/numéroEnregistrement&gt;
    &lt;/société&gt;
    &lt;société&gt;
        &lt;nomSociété&gt;La Truelle S.P.R.L.&lt;/nomSociété&gt;
        &lt;adresse&gt;
            &lt;rue&gt;place de la construction&lt;/rue&gt;
            &lt;numéro&gt;32&lt;/numéro&gt;
            &lt;codePostal&gt;4000&lt;/codePostal&gt;
            &lt;ville&gt;Liège&lt;/ville&gt;
        &lt;/adresse&gt;
    &lt;/société&gt;
&lt;/listSoc&gt;</pre>
```

```

<contact>
    <nom>Pirandello</nom>
    <prenom>Luigi</prenom>
    <fonction>Contremaitre</fonction>
    <telephone>0497 337599</telephone>
</contact>
<numéroEnregistrement ONSS="SUR000" interne="Louche">500/220/40520
</numéroEnregistrement>
</société>
</listSoc>

```

et un fichier xsl (société.xsl), qui définit les transformations à effectuer pour générer une page HTML – nous pouvons générer une base avec NetBeans :



pour obtenir :

```

société.xsl (code généré)
<?xml version="1.0" encoding="UTF-8" ?>

<!--
Document : société.xsl
Created on : 22 août 2007, 17:21
Author   : Vilvens
Description: Purpose of transformation follows.
-->

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
    <xsl:output method="html"/>

    <!-- TODO customize transformation rules
        syntax recommendation http://www.w3.org/TR/xslt
    -->
    <xsl:template match="/">
        <html>
            <head>
                <title>société.xsl</title>

```

```
</head>
<body>
</body>
</html>
</xsl:template>
</xsl:stylesheet>
```

Après modifications, cela donne :

société.xsl (code modifié)

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">

<xsl:template name="saut">
<xsl:text disable-output-escaping="yes">&lt;p&gt;</xsl:text>
</xsl:template>

<xsl:output method="html" encoding="ISO-8859-1" />

<xsl:template match="/">
<html>
<body>

<xsl:for-each select="listSoc/société">
Société associée : <h3><xsl:value-of select="nomSociété"/>*** </h3>
<xsl:text disable-output-escaping="yes">&lt;p&gt;</xsl:text>
Nom du contact : <b><xsl:value-of select="contact/nom"/></b>
<xsl:call-template name="saut"></xsl:call-template>

<xsl:for-each select="contact/email">
Son e-mail : <xsl:value-of select=". "/>
</xsl:for-each>

<xsl:call-template name="saut"></xsl:call-template>

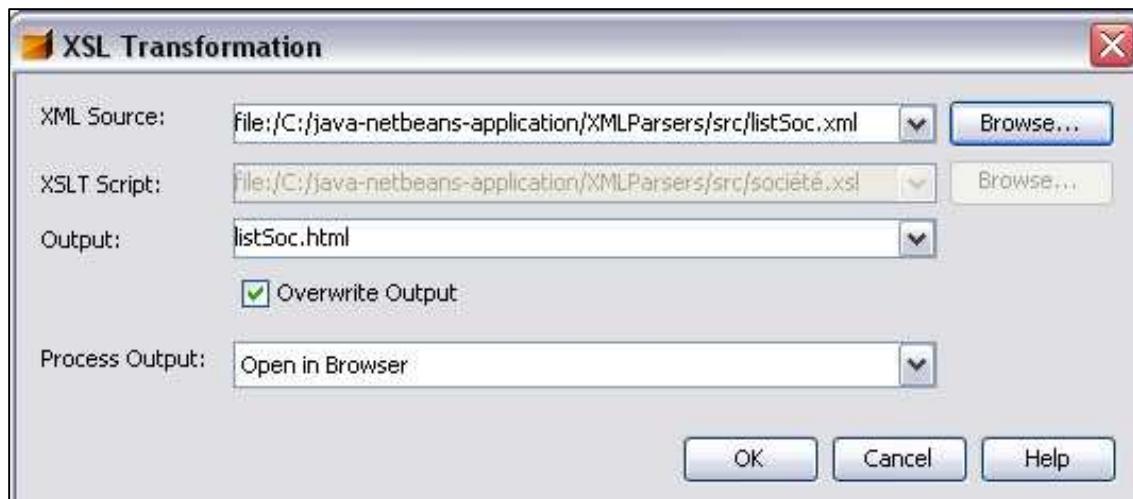
<xsl:for-each select="contact/telephone">
Son numéro de téléphone : <xsl:value-of select=". "/>
</xsl:for-each>

<xsl:call-template name="saut"></xsl:call-template>
-----
<xsl:call-template name="saut"></xsl:call-template>
</xsl:for-each>

</body>
</html>
</xsl:template>

</xsl:stylesheet>
```

Il est possible d'appliquer directement la transformation en choisissant "XSL Transformation" dans le menu contextuel obtenu par un clic droit sur le nom du fichier xsl - on obtient une boîte de dialogue dans laquelle il suffit de choisir le fichier XML auquel la transformation doit être appliquée :



On obtient immédiatement :

The screenshot shows the Mozilla Firefox browser window displaying the transformed HTML output. The address bar shows the URL: file:///C:/java-n.../src/listSoc.html. The page content is as follows:

Société associée :
Genius S.A.***
Nom du contact : **Bidendum**
Son e-mail : abidend@groumf.com Son e-mail : albert@sexy.net

Société associée :
La Truelle S.P.R.L.***
Nom du contact : **Pirandello**
Son numéro de téléphone : 0497 337599

Terminé

En fait, NetBeans utilise les services d'un processeur XSLT nommé **Xalan** qui se trouve dans le fichier xalan.jar (répertoire C:\Program Files\.netbeans-5.5\rave2.0\modules\ext) et dont le

processus reçoit en -IN le nom du fichier xml et en -XSL le nom du fichier xsl. On peut d'ailleurs voir dans la console :

```
| Using com.sun.org.apache.xalan.internal.xsltc.trax.TransformerImpl - default JRE XSLT
| processor.
```

11. Un exemple de transformation XSLT programmée

11.1 De nouveaux packages

Qu'en est-il à présent de la programmation de ce genre de transformations ? Les APIs Java disponibles permettent de prendre en charge des opérations comme celles décrites ci-dessus. Ces APIs **TrAX** (Transformation API for XML) se trouvent dans les packages suivants :

javax.xml.transform	: La classe Transformer qui réalise la transformation et la factory associée pour obtenir un tel objet.
javax.xml.transform.dom	: Les classes DOMSource et DOMResult pour des transformations dont la source et le résultat sont de type DOM.
javax.xml.transform.sax	: Idem, mais pour SAX.
javax.xml.transform.stream	: Idem pour une source ou un résultat sous forme de flux.

En pratique, nous utiliserons les classes fournies par Xerces dans le fichier xalan.jar – ce dernier sera donc monté dans notre projet en plus de jaxp.jar.

11.2 Obtenir et utiliser un transformer

Reprendons donc nos fichiers de l'exemple ci-dessus et tentons de programmer la même transformation au moyen des APIs Java.

Ce dont nous avons besoin est un objet qui instancie la classe **Transformer** (du package javax.xml.transform). Un tel objet est capable de lire en entrée un fichier XML pour fournir en sortie un autre fichier XML, ou une autre forme de fichier.

Comme d'habitude, pour instancier le transformeur lui-même, il nous faudra en obtenir une instance au moyen d'une factory. La *classe factory* se nomme ici **TransformerFactory** (du package javax.xml.transform) et l'objet factory peut s'obtenir au moyen de la méthode inévitable :

```
public static TransformerFactory newInstance()
    throws TransformerFactoryConfigurationError
```

qui permet d'obtenir une instance de **TransformerFactory**, indépendamment de la manière dont cette instance est effectivement implémentée. En fait, cette méthode de classe va successivement rechercher la classe implémentant la factory

- ◆ dans la propriété du système nommée javax.xml.transform.TransformerFactory;
- ◆ dans le fichier lib/jaxp.properties du JRE;
- ◆ dans les API de service (Services API) qui sont éventuellement définies dans META-INF/services/javax.xml.parsers.TransformerFactory (c'est le cas dans xalan.jar);
- ◆ dans l'instance de DocumentBuilderFactory par défaut pour le système.

La classe **TransformerFactoryConfigurationError** est une classe dérivée d'Error, elle-même dérivée de Throwable. Un objet de cette classe est évidemment lancé si l'instanciation demandée se révèle impossible parce qu'aucune classe instanciable (donc, non abstraite) n'a été trouvée.

Donc, finalement :

```
TransformerFactory tf = TransformerFactory.newInstance();
```

2) Cette classe factory va maintenant nous permettre d'obtenir une instance du transformer attendu, instance de la classe **Transformer**, en utilisant la méthode

```
public abstract Transformer newTransformer(Source source)
    throws TransformerConfigurationException
```

qui réclame comme paramètre la feuille de style XSLT à utiliser (donc, le fichier xsl). Cette feuille est présentée comme une implémentation de l'interface **Source** (déclaré dans le package javax.xml.transform), qui représente indifféremment une source XML ou XSL. La convention est d'utiliser comme Source un objet qui instancie la classe **StreamSource** (package javax.xml.transform.stream) : cette classe est sensée représenter un flux qui respecte le format XML (donc, aussi bien un fichier xsl qu'un fichier xml de données proprement dit). Elle comporte une armée de constructeurs, dont les plus utiles sont sans doute :

```
public StreamSource(java.io.File f)
et
public StreamSource(java.lang.String url)
```

Bien sûr, si le fichier xsl n'est pas trouvé, le transformer ne peut rien faire et l'exception **TransformerConfigurationException** sera lancée (la même sera lancée si des erreurs sont détectées dans les templates du fichier xsl).

La transformation en elle-même est demandée au moyen de la méthode

```
public abstract void transform(Source xmlSource, Result outputTarget)
    throws TransformerException
```

Le premier paramètre représente bien sûr le fichier XML tandis que le deuxième permet de désigner le fichier résultat. Ce dernier est donc une implémentation de l'interface **Result** qui définit (très peu précisément) l'interface d'un objet représentant l'arbre résultant d'une transformation XSLT. On utilise habituellement pour lui un objet instance de la classe **StreamResult** qui sert de dépositaire à un fichier texte ou à balise comme HTML ou XML.

Donc, finalement, si tf est un objet **TransformerFactory** et si les différents noms sont des chaînes de caractères :

```
Transformer t = tf.newTransformer(new StreamSource(new File(nomFeuilleDeStyle)));
t.transform(new StreamSource(nomSource), new StreamResult(nomResultat));
```

11.3 Configurer le transformer

Les propriétés du transformer obtenu ne sont pas encore définies autrement que par défaut. Cela peut souvent suffire, notamment dans notre cas. Mais pour les définir selon des besoins particuliers, il nous faut créer une liste de propriétés (donc, comme d'habitude, un

objet **Properties**) dans laquelle nous allons placer, par exemple, le couple (méthode, html). C'est le rôle de la classe **OutputKeys** (du même package javax.xml.transform) de contenir les constantes permettant de fixer ces propriétés, comme par exemple :

```
public static final java.lang.String METHOD
```

dont on devine que les valeurs possibles sont "xml", "html" ou "text". Une fois la liste de propriétés construite, on peut configurer le transformeur au moyen de sa méthode :

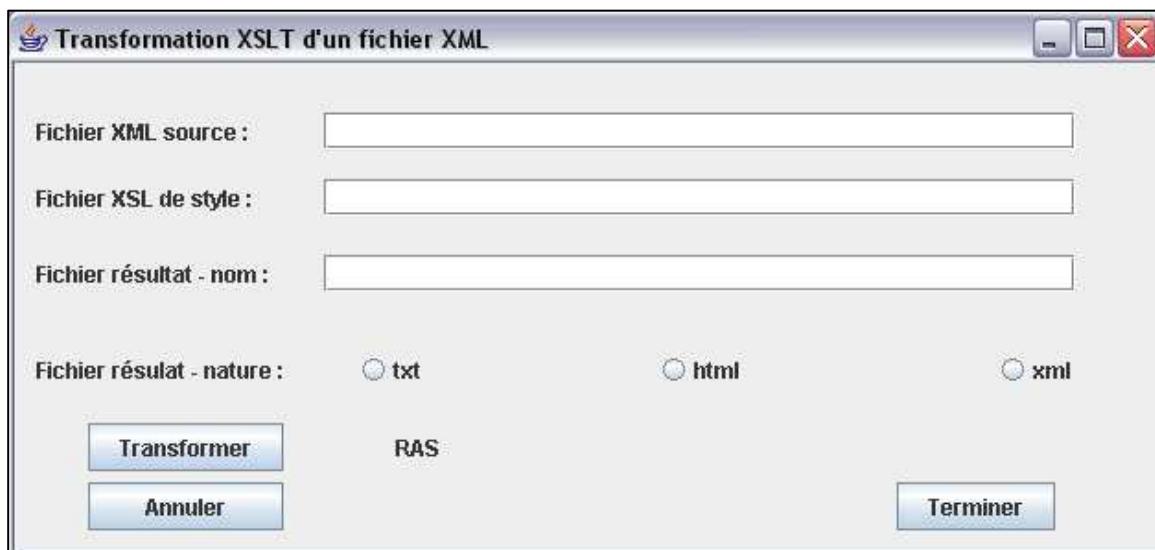
```
public abstract void setOutputProperties(java.util.Properties oformat)
    throws java.lang.IllegalArgumentException
```

On peut donc ajouter à notre code, avant d'appeler la méthode de transformation :

```
Properties prop = new Properties();
prop.put(OutputKeys.METHOD, "html");
t.setOutputProperties(prop);
```

11.4 Un petit programme de transformation

Nous pouvons à présent résumer ce qui précède dans un modeste programme qui réalise une transformation XSLT en usant de l'interface graphique suivant :



Ce programme s'écrit :

```
XslTransform.java
/*
 * XslTransform.java
 * Created on 22 août 2007, 18:08
 */
package parsers;
import java.awt.*;
```

```
import java.io.*;
import java.util.*;

import javax.xml.transform.*;
import javax.xml.transform.stream.*;

/**
 * @author Vilvens
 */

public class XslTransform extends javax.swing.JFrame
{
    public XslTransform() { initComponents(); }
    private void initComponents() { ... }

    private void BTerminerActionPerformed(java.awt.event.ActionEvent evt)
    {
        System.exit(0);
    }

    private void BAnnulerActionPerformed(java.awt.event.ActionEvent evt)
    {
        TFSource.setText("");
        TFXsl.setText("");
        TFResult.setText("");
    }

    private void BTransformerActionPerformed(java.awt.event.ActionEvent evt)
    {
        String nomSource = TFSource.getText();
        String nomFeuilleDeStyle = TFXsl.getText();
        String nomResultat = TFResult.getText();

        System.out.println("Paramètres recueillis : " + nomSource + " --> " +
                           nomFeuilleDeStyle + " -->" + nomResultat);

        System.out.println("Recherche d'une factory de Transformer ...");
        TransformerFactory tf = TransformerFactory.newInstance();
        System.out.println("Factory de Transformer trouvée !");
        try
        {
            System.out.println("Recherche d'un Transformer ..");
            Transformer t = tf.newTransformer(new StreamSource (new File(nomFeuilleDeStyle)));
            System.out.println("Transformer trouvé !");
        }

        String methode = "txt";
        if (CBHtml.isSelected()) methode="html";
        else if (CBXml.isSelected()) methode="xml";
        StringBuffer nomCible = new StringBuffer(nomResultat);
        nomCible.append("." +methode);
```

```

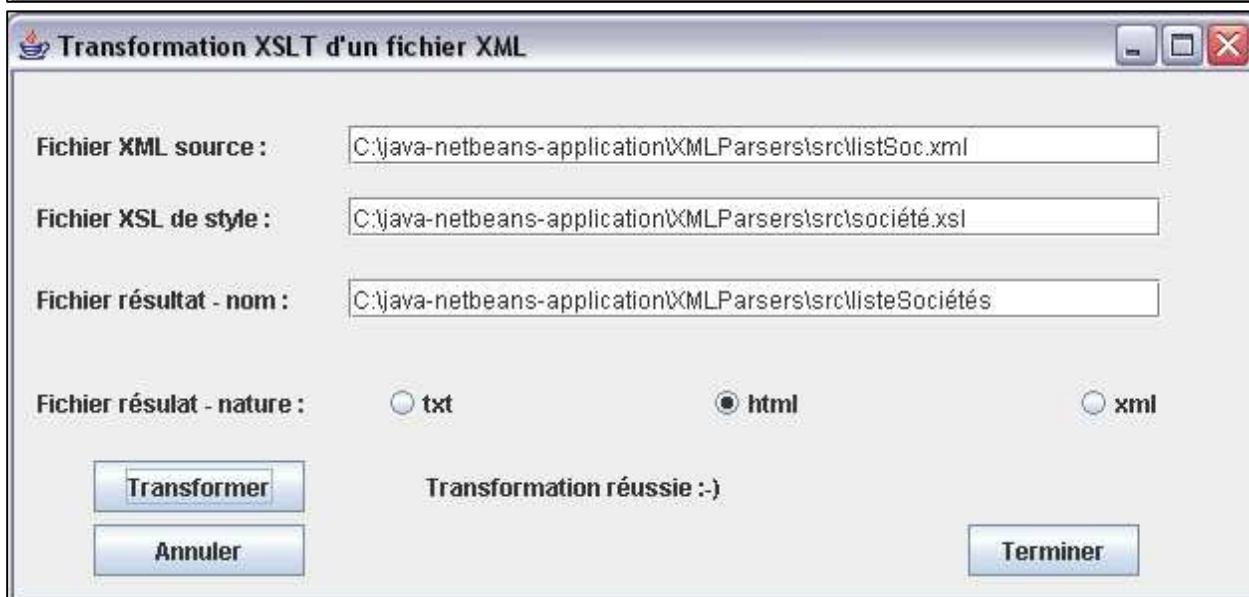
nomResultat = new String(nomCible);
Properties prop = new Properties();
prop.put(OutputKeys.METHOD, methode);
t.setOutputProperties(prop);
System.out.println("Essai de transformation ...");
t.transform(new StreamSource(nomSource), new StreamResult(nomResultat));
System.out.println("Transformation terminée !");
LResultat.setText("Transformation réussie :-) ");
}
catch (TransformerConfigurationException e)
{
    System.out.println("oh oh : pas de transformer ? " + e.getMessage());
}
catch (TransformerException e)
{
    System.out.println("oh oh : problème dans la transformation ? " + e.getMessage());
    LResultat.setText("Transformation ratée :-( ");
}
}

public static void main(String args[])
{
    java.awt.EventQueue.invokeLater(new Runnable()
    {
        public void run()
        {
            new XsltTransform().setVisible(true);
        }
    });
}

private javax.swing.JButton BAnnuler;
private javax.swing.JButton BTerminer;
private javax.swing.JButton BTransformer;
private javax.swing.JRadioButton CBHtml;
private javax.swing.JRadioButton CBTxt;
private javax.swing.JRadioButton CBXml;
private javax.swing.JLabel LResultat;
private javax.swing.JTextField TFRresult;
private javax.swing.JTextField TFSource;
private javax.swing.JTextField TFXsl;
private javax.swing.ButtonGroup buttonGroup1;
...
}

```

L'exécution dans le GUI peut ressembler à ceci :



Le fichier listeSociétés.html a bien été créé, ayant le même aspect que celui de l'exemple du paragraphe précédent.

11.5 La sauvegarde dans un fichier XML (2)

La méthode d'écriture d'un arbre DOM dans un fichier XML qui utilise un **BaseMarkupSerializer** (voir paragraphe 9.2 ci-dessus) est en fait considérée comme obsolète [deprecated] depuis Xerces 2.9.0. Il est à présent recommandé d'utiliser un "DOM Level 3 LSSerializer" ou surtout une API TrAX qui utilise tout simplement un **Transformer**. Celui-ci sera simplement configuré pour produire le fichier souhaité en utilisant principalement les constantes de la classe **OutputKeys** :

```
public static final String ENCODING
public static final String INDENT
public static final String DOCTYPE_SYSTEM
public static final String VERSION
```

(en plus de la constante connue METHOD) dans la méthode de la classe Transformer :

```
public abstract void setOutputProperty(String name, String value)
throws IllegalArgumentException
```

Cela donne donc simplement :

XMLSerialize.java
package xmlserialize;
import javax.xml.transform.*;
import javax.xml.transform.dom.*;
import javax.xml.transform.stream.*;
import org.w3c.dom.Node;
import java.io.FileOutputStream;
public class XMLSerialize
{
public static void main(String[] args)
{
TransformerFactory <i>tf</i> = TransformerFactory.newInstance();
Node documentCree=null;
/* Création du document ...*/
...
/* Fin création du document ...*/
try
{
Transformer <i>t</i> = <i>tf</i> .newTransformer();
<i>t</i> . setOutputProperty (OutputKeys.METHOD, "xml");
<i>t</i> .setOutputProperty(OutputKeys.ENCODING, "ISO-8859-1");
<i>t</i> .setOutputProperty(OutputKeys.INDENT, "yes");
<i>t</i> .setOutputProperty(OutputKeys.DOCTYPE_SYSTEM, "employes.dtd");
<i>t</i> .setOutputProperty(OutputKeys.VERSION, "1.0");
FileOutputStream <i>fos</i> = new FileOutputStream
("c:\\java-netbeans-application\\XmlEncodeDecode\\DomTreeTrans.xml");
<i>t</i> .transform(new DOMSource(documentCree), new StreamResult (<i>fos</i>));
}
catch (Exception e)
{ System.out.println("Erreur de transformation : " + e.getMessage()); }
}
}

12. La sérialisation XML

12.1 Une sérialisation texte

Nous connaissons déjà, et depuis longtemps, le principe de la sérialisation des objets Java, uniquement conditionnée par le fait pour la classe instanciée implémenter l'interface **Serializable**. On qualifie encore ce processus, sur lequel nous ne reviendrons pas, de "sérialisation binaire" – car il existe une "sérialisation texte" ou plutôt "xml". Il existe effet deux classes **XMLEncoder** et **XMLDecoder** qui se proposent respectivement de créer (à partir d'un objet) et de décoder (pour reconstruire un objet) un document XML décrivant l'état de l'objet en question. A priori, cependant, l'objet est supposé respecter le paradigme des Java Beans (donc notamment être caractérisé par des propriétés et posséder un constructeur par défaut) – encore que ceci puisse être contourné. Le fichier XML en lui-même est en fait un descriptif de l'état de l'objet, une espèce de ***script de travail*** à utiliser par le **XMLDecoder** pour reconstituer l'objet en mémoire. Abordons immédiatement un exemple simple en reprenant un Java Bean extrait de Java (I), le célèbre AlertBean :

AlertBean.java

```
package xmlencodeddecode;

import java.util.*;
import java.io.*;
import java.beans.*;

public class AlertBean implements Serializable
{
    private int multipleDeclenchement;
    private int niveauxAlerte;
    private boolean enMarche;

    final int multipleDeclenchementParDefaut = 5;
    static final int TROP_ALERTES = 10;
    private Vector alertListeners;
    protected PropertyChangeSupport GestProp = new PropertyChangeSupport(this);

    public AlertBean ()
    {
        multipleDeclenchement = multipleDeclenchementParDefaut;
        niveauxAlerte = 3; enMarche = false; alertListeners = new Vector();
    }

    public AlertBean (int mDecl, int na)
    {
        multipleDeclenchement = mDecl; niveauxAlerte = na;
        enMarche = false; alertListeners = new Vector();
    }

    public int getPeriode() { return multipleDeclenchement; }
    public void setPeriode(int m) { multipleDeclenchement = m; }
    public int getNiveauxAlerte() { return niveauxAlerte; }
    public void setNiveauxAlerte (int na) { niveauxAlerte = na; }
    public boolean isEnMarche() { return enMarche; }
```

```
public void setEnMarche(boolean e)
{
    boolean ancienEtatDeMarche = enMarche; enMarche = e;
    GestProp.firePropertyChange("enMarche",
        new Boolean(ancienEtatDeMarche), new Boolean(e));
}

public void init() { ... }
public void stop(){ ... }
public void run(){ ... }
public void addAlertListener (AlertListener al) { ... }
public void removeAlertListener (AlertListener al) { ... }
protected void notifyAlertDetected(int niv) { ... }
public void addPropertyChangeListener(PropertyChangeListener l) { ... }
public void removePropertyChangeListener(PropertyChangeListener l) { ... }
}
```

La classe **XMLEncoder** possède un constructeur réclame le flux sur lequel sera écrit le fichier XML :

public **XMLEncoder**(OutputStream out)

Sa méthode

public void **writeObject**(Object o)

permet de réaliser la "sérialisation XML" tandis que

public void **close()**

permet de refermer le flux utilisé. Sans utres commentaires, on peut donc programmer ceci :

AlertBeanXMLSerialize.java

```
package xmlencodeddecode;

import java.beans.*;
import java.io.*;
import java.util.logging.Level;
import java.util.logging.Logger;

/**
 * @author VILVENS
 */

public class AlertBeanXMLSerialize
{
    public static void main(String[] args)
    {
        AlertBean ab = null;
```

```

try
{
    ab = (AlertBean) Beans.instantiate(null,"xmlencodeddecode.AlertBean");
    System.out .println("main> AlertBean instancié");
}
catch (ClassNotFoundException e)
{ System.out .println("Classe non trouvée"); System.exit(0); }
catch (IOException e)
{ System.out .println("Fichier de sérialisation non trouvé"); System.exit(0); }
ab.setPeriode(7);
ab.setNiveauxAlerte(6);
ab.setEnMarche(true);

BeanInfo info=null;
try
{
    info = Introspector.getBeanInfo(AlertBean.class);
}
catch (IntrospectionException ex) {... }
PropertyDescriptor[] propertyDescriptors = info.getPropertyDescriptors();
for (int i = 0; i < propertyDescriptors.length; ++i)
{
    PropertyDescriptor pd = propertyDescriptors[i];
    System.out.println(pd.getName());
}

XMLEncoder e=null;
try
{
    e = new XMLEncoder(new BufferedOutputStream(
        new FileOutputStream("AlertBeanNew.xml")));
}
catch (FileNotFoundException ex)
{ System.out.println("Problème de XMLEncoder : "+ ex); }
e.writeObject(ab);
e.close();
}
}

```

Le résultat sera le fichier suivant :

AlertBeanNew.xml

```

<?xml version="1.0" encoding="UTF-8"?>

<java version="1.6.0_13" class="java.beans.XMLDecoder">
<object class="xmlencodeddecode.AlertBean">
<void property="enMarche">
<boolean>true</boolean>
</void>

```

```
<void property="niveauxAlerte">
<int>6</int>
</void>
<void property="periode">
<int>7</int>
</void>
</object>
</java>
```

Très clair ...

La console quant à elle affichera :

```
main> AlertBean instancié
0. class
1. enMarche
2. niveauxAlerte
3. periode
```

Ceci provient de l'utilisation de la classe **Introspector** (package java.beans) qui mérite bien son nom puisqu'elle possède la méthode :

```
public static BeanInfo getBeanInfo(Class<?> beanClass) throws IntrospectionException
```

Le BeanInfo ainsi récupéré permet d'obtenir, entre autres, une liste de descripteurs de propriétés au moyen de la méthode :

```
PropertyDescriptor [] getPropertyDescriptors()
```

Un objet **PropertyDescriptor** permet de tout savoir d'une propriété d'un bean, comme :

```
public String getNameisBound()
public boolean isConstrained()
```

L'opération de désérialisation est du ressort de la classe **XMLDecoder** dont le constructeur réclame le flux fournissant le texte XML décrivant le bean :

```
public XMLDecoder(InputStream out)
```

La méthode

```
public Object readObject()
```

permet évidemment de récupérer les propriétés du bean. On peut donc écrire :

AlertBeanXMLDeserialize.java

```

package xmlencodeddecode;

import java.beans.*;
import java.io.*;
import java.util.logging.Level;
import java.util.logging.Logger;

/**
 * @author VILVENS
 */
public class AlertBeanXMLDeserialize
{
    public static void main(String[] args)
    {
        XMLDecoder d=null;
        AlertBean result = null;
        try
        {
            d = new XMLDecoder(new BufferedInputStream(
                new FileInputStream("AlertBeanNew.xml")));
            result = (AlertBean)d.readObject();
        }
        catch (FileNotFoundException ex)
        {
            System.out.println("Problème de XMLDecode : " + ex);
        }
        d.close();
        System.out.println("Bean récupéré :");
        System.out.println("période = " + result.getPeriode());
        System.out.println("niveaux alerte = " + result.getNiveauxAlerte());
        System.out.println("en marche ? = " + result.isEnMarche());    }
    }
}

```

Si on a, au préalable, modifié des valeurs dans le fichier XML, on obtiendra sur la console :

Bean récupéré :
 période = 15
 niveaux alerte = 3
 en marche ? = true

On aura compris que l'on dispose donc ici d'un outil d'échange d'objets entre plates-formes doté de qualités évidentes :

- ◆ portabilité : le fichier XML dit "comment" reproduire l'objet, indépendamment des versions des classes et JVMs des machines où l'on travaille;
- ◆ structure compacte : les valeurs par défaut ne sont pas recopiées dans le fichier XML;
- ◆ tolérance aux erreurs : tout comme nous l'avions déjà constaté dans les parsings SAX, une erreur dans une partie du fichier n'implique que la perte de la zone qui la contient, pas de tout l'arbre XML.

Remarque

Il est possible de "modeler" cette persistance, par exemple pour ne sérialiser que certaines propriétés, en définissant un PersistenceDelegate

```
public PersistenceDelegate getPersistenceDelegate(Class<?> type)
```

12.2 Les éléments syntaxiques d'un document XMLEncoder/Decoder

Intéressons-nous plus précisément à la structure des fichiers XML créés par un XMLEncoder et utilisés par un XMLDecoder.

a) Comme tout fichier XML, on trouve d'abord la ligne de prologue (<?xml ...>) suivie d'une ligne spécifique à Java : le tag <java> indique la version du JDK et la classe de décodage pour laquelle le document XML a été créé :

```
<?xml version="1.0" encoding="UTF-8" ?>
<java version="1.4.0" class="java.beans.XMLDecoder">
```

L'anti-tag </java> referme le document.

b) Le tag <object> permet de déclarer et de configurer un objet : il contient d'autres tags dont le rôle est d'indiquer soit comment créer l'objet (on parle d'"**expression**") soit une instruction à exécuter dont l'effet se répercutera sur l'état de l'objet (on parle de "**statement**") – par exemple :

```
<object class="javax.swing.JFrame">
...
</object>
```

c) Les expressions sont donc des appels de méthodes qui vont renvoyer une valeur : elles sont donc représentées par de nouveaux tags <object method=...>, les arguments de la méthode étant cités à l'intérieur par des tags <string>, <int>, etc .. - par exemple :

```
<object class="javax.swing.JButton" method="new">
  <string>OK</string>
</object>
```

d) Les statements sont des appels de méthodes qui ne retournent rien : elles sont donc représentées par des tags <void method= ...>. Leur action portera sur un objet qui est celui dont le tag <object> entoure le statement - par exemple :

```
<object class="javax.swing.JButton">
  <void method="setText">
    <string>Continuer</string>
  </void>
</object>
```

e) Il peut arriver qu'un même objet soit utilisé en plusieurs endroits du script XML. Dans ce cas,

- ♦ on lui donne un identifiant qui permettra de la désigner de manière univoque dans le document en utilisant l'attribut "id" – par exemple :

```
<object id="button1" class="javax.swing.JButton"/>
```

- ♦ on y fait référence en utilisant l'attribut "idref" – par exemple :

```
<object idref="button1"/>
```

Un cas particulier de l'utilisation de id est par exemple celui-ci :

```
<object class="java.util.Date">
  <void id="now" method="getTime"/>
</object>
```

- on donne ainsi un identifiant à une expression (ou plutôt son résultat).

- f) L'attribut "property" du tag `<void>` permet de fixer la valeur d'une propriété plus clairement qu'en utilisant un statement – par exemple :

```
<object class="javax.swing.JButton">
  <void property="label">
    <string>Hello</string>
  </void>
</object>
```

- g) Il est même possible de définir des tableaux au moyen du tag array – par exemple

```
<array class="java.lang.String" length="3">
  <void index="1">
    <string>Hello, world</string>
  </void>
</array>
```

crée un tableau de 3 chaînes de caractères tandis que

```
<array class="int">
  <int>123</int>
  <int>456</int>
</array>
```

crée un tableau de 2 entiers.

12.3 Un exemple démonstratif

Le javadoc de la classe XMLEncoder fournit l'exemple suivant :

FenetreBean.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.0" class="java.beans.XMLDecoder">
<object class="javax.swing.JFrame">
<void property="name">
<string>frame1</string>
</void>
<void property="bounds">
<object class="java.awt.Rectangle">
<int>0</int>
<int>0</int>
<int>400</int>
<int>200</int>
</object>
</void>
<void property="title">
<string>Exemple de désérialisation</string>
</void>
<void property="contentPane">
<void method="add">
<object class="javax.swing.JButton">
<void property="label">
<string>Bonjour à tous !!!</string>
</void>
</object>
</void>
</void>
<void property="visible">
<boolean>true</boolean>
</void>
</object>
</java>
```

On peut donc imaginer le petit programme suivant :

XMLDecodeFenetre.java

```
package xmlencodeddecode;

import java.beans.*;
import java.io.*;

/**
 * @author Vilvens
 */
public class XMLDecodeFenetre
{
    public static void main(String[] args)
    {
        XMLDecoder d=null;
```

```
try
{
    d = new XMLDecoder(new BufferedInputStream(
        new FileInputStream("FenetreBean.xml")));
    Object result = d.readObject();
}
catch (FileNotFoundException ex)
{
    System.out.println("Problème de XMLDecode : " + ex);
}
d.close();
}
```

Résultat :



12.4 La structure d'un document XMLEncoder/Decoder

Terminons de manière rigoureuse : la **DTD** à laquelle les classes XMLEncoder et XMLDecoder se réfèrent est la suivante.

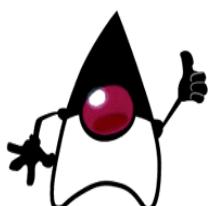
```
<!ELEMENT java ( object | void | string | class | null | array | boolean |
    byte | char | short | int | long | float | double )*>
<!ATTLIST java
    version CDATA #IMPLIED
    class   CDATA #IMPLIED
>

<!ELEMENT boolean (#PCDATA)>
<!ELEMENT byte   (#PCDATA)>
<!ELEMENT char   (#PCDATA)>
<!ELEMENT short  (#PCDATA)>
<!ELEMENT int    (#PCDATA)>
<!ELEMENT long   (#PCDATA)>
<!ELEMENT float  (#PCDATA)>
<!ELEMENT double (#PCDATA)>
<!ELEMENT string (#PCDATA)>
<!ELEMENT class  (#PCDATA)>
<!ELEMENT null   (#PCDATA)>
```

```
<!ELEMENT object ( object | void | string | class | null | array | boolean |
    byte | char | short | int | long | float | double )*>
<!ATTLIST object
    id    ID    #IMPLIED
    idref IDREF #IMPLIED
    class CDATA #IMPLIED
    field CDATA #IMPLIED
    method CDATA #IMPLIED
    property CDATA #IMPLIED
    index  CDATA #IMPLIED
>

<!ELEMENT array ( object | void | string | class | null | array | boolean |
    byte | char | short | int | long | float | double )*>
<!ATTLIST array
    id    ID    #IMPLIED
    class CDATA #IMPLIED
    length CDATA #IMPLIED
>

<!ELEMENT void ( object | void | string | class | null | array |
    boolean | byte | char | short | int | long | float | double )*>
<!ATTLIST void
    id    ID    #IMPLIED
    class CDATA #IMPLIED
    method CDATA #IMPLIED
    property CDATA #IMPLIED
    index  CDATA #IMPLIED
>
```



Il est donc possible de sauver des objets Java dans un fichier XML et, à l'inverse, de créer des objets Java à partir d'un fichier XML : on parle encore parfois de marshalling et unmarshalling. On pressent ici la pointe d'un iceberg qui vogue vers les échanges multi-platerformes des systèmes distribués et des services Web ...

Voici juste une toute petite introduction ...

XIX. Un panorama général de J2EE

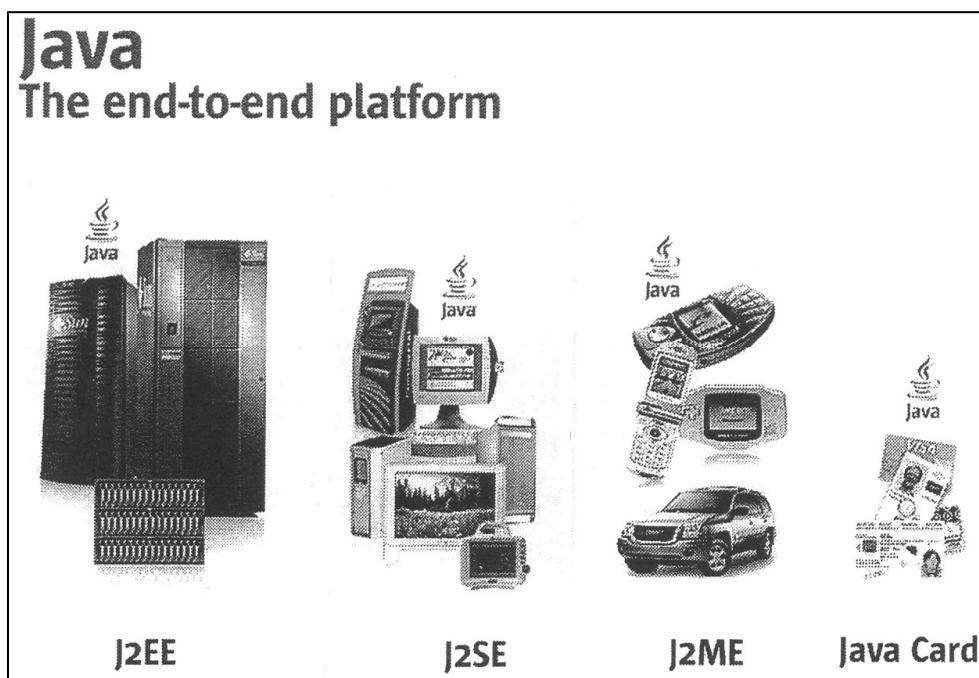


Une longue expérience m'a démontré que le mariage, pourtant classique, de l'obscurantisme et de l'autorité donne rarement pour fruit une compréhension lumineuse des problèmes de la vie.

(Greg, Achille Talon)

1. Les quatre plates-formes Java de Sun

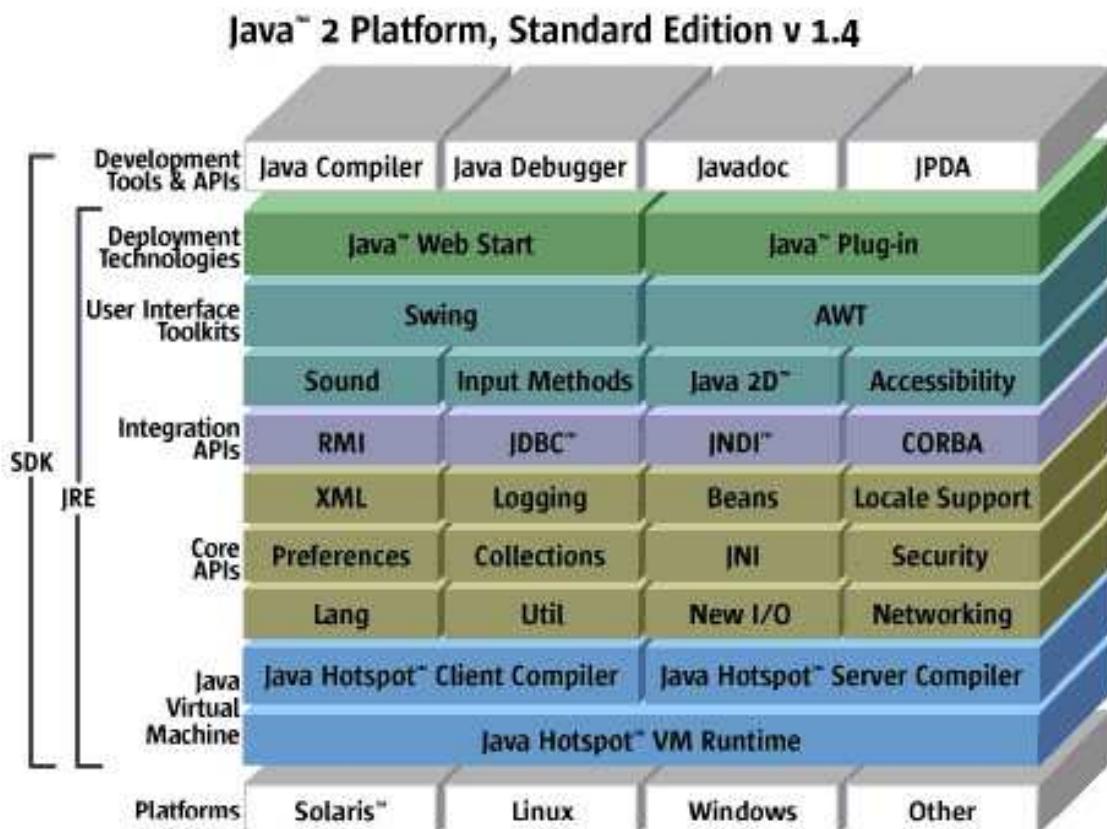
Sun propose actuellement 4 plates-formes de développement Java,



chacune orientée vers un secteur bien précis :

- a) la plate-forme **Java Card** est évidemment destinée au développement d'applications sur cartes à puces¹⁴;
- b) la plate-forme **J2ME** (**Java 2 Micro Edition**) est destinée au développement d'applications destinées aux mobiles et aux systèmes embarqués¹;
- c) on ne présente plus vraiment le **J2SE** (**Java 2 Standard Edition**), destinée au développement d'**applications classiques** et d'**applications Web**; sa structure se reflète dans le schéma suivant (honteusement copié chez Sun ;-)) :

¹⁴ voir "Langage Java (IV) : Programmation de protocoles applicatifs et de technologies d'e-commerce" – du même auteur ...



d) enfin, la plate-forme **J2EE** (Java 2 Enterprise Edition) est destinée au développement d'applications distribuées et architecturées en multi-tiers – c'est ici que des technologies comme RMI et les EJBs interviennent.

2. Le principe et l'architecture de RMI

RMI (Remote Method Invocation) est un mécanisme de Java qui permet aux objets locaux d'une application d'interagir avec un objet instancié sur une machine virtuelle distante, reliée à la VM locale par un réseau. Il est ainsi possible d'utiliser les méthodes de cet objet distant comme s'il était local, c'est-à-dire comme s'il était utilisé sur la machine virtuelle locale. Cette architecture est historiquement à la base des technologies Java orientées vers les systèmes distribués.

L'architecture qui se met ainsi en place est celle d'un objet **serveur**, qui met certaines de ses méthodes à la disposition d'autres objets qui se trouvent sur le réseau commun, objets qui sont les **clients** implicites de ce serveur et qui rechercheront une instance de ce serveur en cas de nécessité.

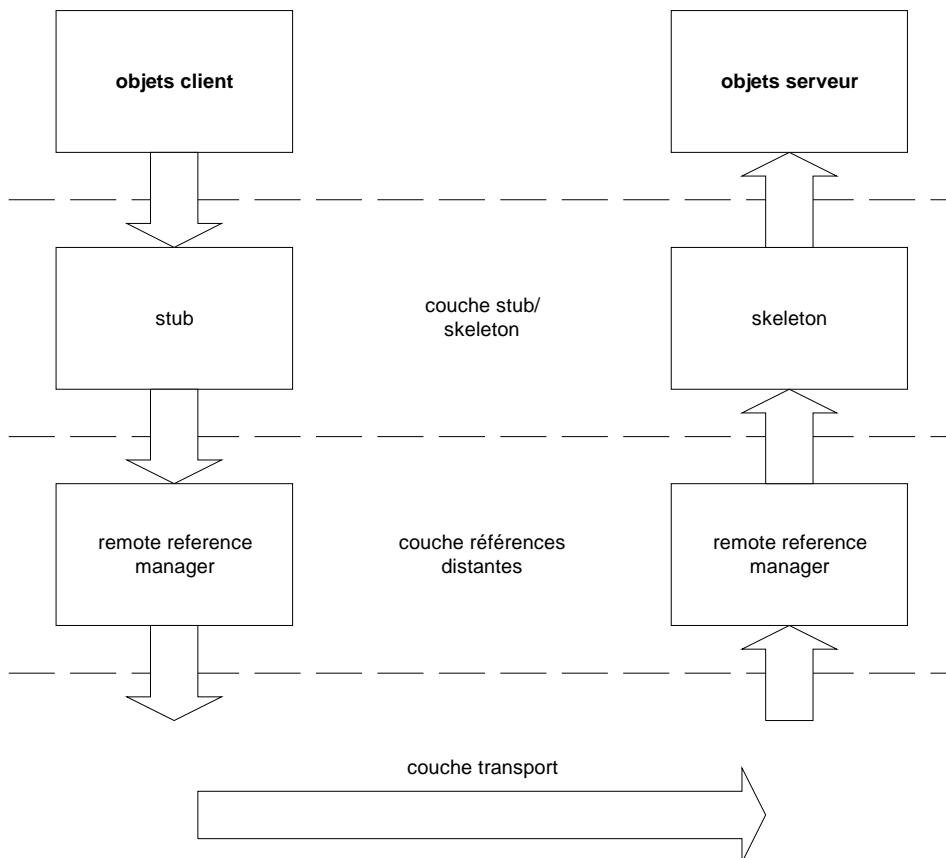
Clairement, nous sommes dans le contexte de ce que l'on appelle les "**objets distribués**". En fait, on passe ainsi de la méthode classique de communication réseau, qui consiste à envoyer des données sur le réseau en utilisant un protocole donné (par exemple, TCP), à une méthode qui consiste à exporter un objet vers la plate-forme distante, ses méthodes devenant implicitement l'implémentation du protocole de communication.

On peut encore remarquer que RMI est "**Java-only**", puisque les seuls objets distants utilisables sont ceux programmés en Java. CORBA¹⁵ est indépendant du langage et lève donc cette restriction, en payant évidemment le prix d'une plus grande complexité.

L'architecture réseau de RMI se compose de 3 couches :

- ◆ la couche stub/skeleton : elle fournit l'interface que le client et le serveur vont utiliser pour communiquer; on a coutume encore de les considérer comme des proxys de communication; plus précisément :
 - le **stub** est un proxy local, c'est-à-dire un programme servant d'interface sur la machine locale avec l'objet distant;
 - le **skeleton** est un proxy distant, c'est-à-dire un programme servant d'interface sur la machine distante pour invoquer l'objet serveur;
- ◆ la couche de référence distante : elle établit la jonction entre les proxys et la couche de transport de bas niveau; elle gère les références sur les objets distants;
- ◆ la couche transport : elle véhicule les requêtes vers l'objet distant.

Schématiquement :



Le client invoque une méthode distante en invoquant en fait une méthode du stub concerné. Celui-ci comporte une référence interne vers l'objet distant (en clair, il utilise une connexion

¹⁵ voir les cours de T.Bastianelli à ce sujet

TCP/IP vers la machine distante contenant l'objet distant référencé); cette référence va évidemment lui permettre de communiquer la requête vers l'objet distant à la couche de référence distante. Cependant, il faut convertir les paramètres de la méthode en un format portable qui pourra effectivement être transmis à l'objet distant. Cette opération est appelée le **marshalling** et fournit une version sérialisée des paramètres; il en sera du moins ainsi si l'objet distant visé implémente l'interface **Remote** (voir paragraphe suivant) – sinon, c'est le résultat d'une simple sérialisation en bytes qui sera envoyée. On devine sans peine la suite : envoi de l'invocation ainsi sérialisée au bas niveau, unmarshalling des paramètres, appel de la méthode de l'objet serveur.

Et bien sûr, si la méthode prévoit un retour de valeurs, celles-ci suivent le cheminement inverse. On parle encore de "**boucle RMI**".

Remarque

En Java 2 (JDK 1.2 et suivants), *le skeleton n'existe plus* : le stub communique, par le réseau, directement avec l'objet distant.

3. **Un interface d'objet distant**

Tout objet distant permet donc d'invoquer ses méthodes pour les faire fonctionner sur une machine virtuelle qui n'est pas sa machine virtuelle locale. Pour que cela soit possible, sa classe doit implémenter, directement ou pas, un interface particulier, appelé **Remote** et défini dans le package **java.rmi** (lequel possède d'ailleurs un certain nombre de sous-packages), qui indique que les méthodes citées seront invoquées à distance. Cet interface Remote ne comporte en lui-même aucune méthode : il joue un rôle analogue, par exemple, à Serializable, en ce sens qu'il indique simplement une caractéristique particulière de la classe qui l'"implémente".

Le plus souvent, on commence par créer un interface qui sera implémenté par notre classe distante. Notre interface dérivera donc de Remote. Il définit en fait le protocole d'utilisation de l'objet distant.

L'interface du serveur distant et son implémentation feront partie d'un package AccesRessourceServer. Plus précisément, l'interface sera créé dans un sous-package Interface tandis que l'implémentation le sera dans le sous-package Implementation. Pourquoi cette séparation ? Parce seul le sous-package Interface sera nécessaire au client ...

Pour notre exemple, on peut donc imaginer l'interface AccesRessourceServerInter :

AccesRessourceServerInter.java
package AccesRessourceServer.Interface;
import java.rmi.*;
public interface AccesRessourceServerInter extends Remote
{
String getMessage() throws RemoteException;
void setEnService(String n) throws RemoteException;
void setALARret(String n) throws RemoteException;
boolean setConnexion (String n) throws RemoteException;
boolean closeConnexion (String n) throws RemoteException;
int getNombreClientsConnectés () throws RemoteException;
boolean isEnService () throws RemoteException;

```

public Object getNextInfo() throws RemoteException;
public int getNombreInfosDisponibles() throws RemoteException;
}

```

On remarquera l'exception **RemoteException** (également définie dans le package java.rmi) qui est évidemment lancée dès qu'un problème réseau se fait jour.

4. La classe de l'objet serveur distant

La classe d'un objet distant utilisant TCP pour la couche transport hérite habituellement (c'est-à-dire si l'objet serveur doit être créé dès le lancement de l'application et pas à la demande) de la classe **UnicastRemoteObject** : celle-ci fournit des méthodes hashCode, equals et toString adaptées aux objets distants. Notre classe devra donc tout d'abord

- ◆ hériter de UnicastRemoteObject;
- ◆ implémenter AccesRessourceServerInter;

ce qui implique la définition d'un constructeur et des méthodes de l'interface. Comme notre serveur sera connu par l'interface qu'il implémente, il n'aura donc comme méthodes publiques que celles qui font partie de l'interface. Notre classe s'écrit donc finalement :

```

AccessRessourceServerClasse.java
package AccesRessourceServer.Implementation;

import java.rmi.*;
import java.rmi.server.*;
import java.util.*;

public class AccessRessourceServerClasse extends UnicastRemoteObject implements
AccesRessourceServerInter
{
    private boolean enService;
    private String messageCourant;
    private Vector users, usersConnectés;
    private Vector citations;
    private int nbreCitations;
    private int posElementCourant = 0;

    public AccessRessourceServerClasse()throws RemoteException
    {
        super();
        enService=false;
        users = new Vector();
        users.addElement("Vilvens");users.addElement("Charlet");users.addElement("Merce");
        usersConnectés = new Vector();
        messageCourant = new String("--- Access Ressource Server (c) ---");
        citations = new Vector();
        citations.addElement("Qui a coulé le Titanic ? Iceberg, encore un Juif !");
    }
}

```

```
citations.addElement("Qui n'a pas connu James n'a pas connu l'Amour");
citations.addElement("Qui veut voyager loin ménage sa monture");
nbreCitations = citations.size();
}
public String getMessage() throws RemoteException { return messageCourant; }
public void setEnService(String n) throws RemoteException
{
    if (users.contains(n))
    {
        enService=true;
        messageCourant = new String ("--- Serveur mis en marche ---");
    }
}
public void setALARret(String n) throws RemoteException
{
    if (users.contains(n))
    {
        enService=false;
        messageCourant = new String ("--- Serveur mis à l'arret ---");
    }
}
public boolean setConnexion (String n) throws RemoteException
{
    if (! usersConnectés.contains(n) && enService)
    {
        usersConnectés.addElement(n);
        messageCourant = new String ("--- Nouvelle connexion acceptée ---");
        return true;
    }
    else
    {
        messageCourant = new String ("--- Une seule connexion par utilisateur acceptée ---");
        return false;
    }
}
public boolean closeConnexion (String n) throws RemoteException
{
    if (! usersConnectés.contains(n) && enService)
    {
        usersConnectés.removeElement(n);
        messageCourant = new String ("--- Connexion supprimée ---");
        return true;
    }
    else return false;
}
public int getNombreClientsConnectés () throws RemoteException
{
    return usersConnectés.size();
}
```

```

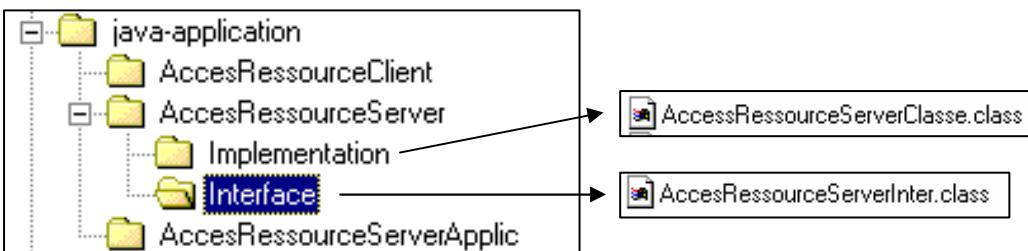
public boolean isEnService () throws RemoteException
{
    return enService;
}
public Object getNextInfo() throws RemoteException
{
    if (enService)
    {
        messageCourant = new String ("--- Une information fournie ---");
        int index=posElementCourant;
        posElementCourant++; if (posElementCourant==nbreCitations) posElementCourant=0;
        return citations.elementAt(index);
    }
    else return new String ("--- NON ---");
}
public int getNombreInfosDisponibles() throws RemoteException
{
    return nbreCitations;
}
}

```

On remarquera la méthode *getNextInfo()* qui, dans l'interface, est déclarée renoyer un Object(), c'est-à-dire n'importe quoi, et qui renvoie ici en fait un String.

5. Un stub et un skeleton

A ce stade, nous disposons, après compilation, de deux fichiers AccesRessourceServerInter.class et AccessRessourceServerClasse.class qui se trouvent respectivement dans les sous-répertoires Interface et Implementation du répertoire qui porte le nom du package, soit AccesRessourceServer. On supposera que le répertoire de base est java-application.

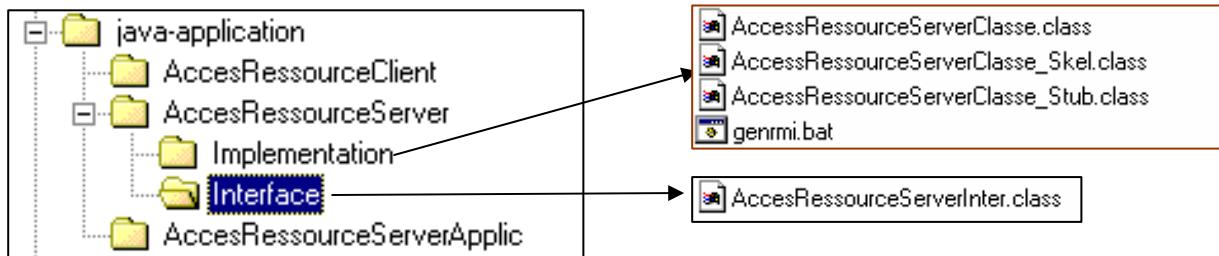


Il faut à présent générer le stub et le skeleton. Cette génération s'obtient en utilisant un compilateur rmi, appelé **rmic** et qui se trouve dans le répertoire bin du JDK. Dans notre cas, on peut donc commander depuis une fenêtre DOS :

```
c:> set PATH=C:\Program Files\Java\jdk1.6.0_13\bin;%PATH%
```

```
c:\java-application\AccesRessourceServer\Implementation>
rmic -classpath c:\java-application; C:\Program Files\Java\jdk1.6.0_13\lib\classes.zip
AccesRessourceServer.Implementation.AccessRessourceServerClasse
ce qui aura pour résultat la création des deux fichiers
```

AccessRessourceServerClasse_Stub.class et **AccessRessourceServerClasse_Skel.class** qui se trouvent dans le répertoire Implementation. Inutile de dire qu'un fichier batch (disons genrmi.bat) s'impose ici ...



En fait, comme déjà dit, le squelette n'est plus nécessaire dans la version 1.2 de RMI. Cependant, rmic continue à le générer pour rester compatible avec des clients RMI 1.1 (rmic admet le commutateur `-v1.2` pour se restreindre à cette version 1.2 et ne plus générer de squelette).

6. Le serveur proprement dit

L'objet serveur va réaliser plusieurs tâches.

Tout d'abord, les seules classes qui peuvent être chargées par le chargeur de classe d'une application (nous ne parlons pas ici d'une applet, à la sécurité plus complexe) sont celles que l'on peut trouver en utilisant le CLASSPATH sur le système local. Ce qui est, bien entendu, ennuyeux quand on a l'intention d'utiliser un objet distant. Il nous faut donc instancier un gestionnaire de sécurité particulier qui nous permettra malgré tout d'utiliser ce code distant. Ce gestionnaire peut être un objet instanciant la classe **RMISecurityManager**. Celle-ci implémente un "security manager" pour applications, encore plus agressif que celui des applets : il ne permet que la définition et l'accès aux méthodes de la classe; bien sûr, il est toujours possible de définir un surveillant moins méfiant en utilisant une classe dérivée de RMISecurityManager. D'une manière ou d'une autre, ce responsable de la sécurité doit exister, sans quoi RMI ne permettra l'instanciation que des seules classes connues dans le CLASSPATH de l'application cliente ...

Ensuite, il faut créer une instance de la classe dont les méthodes seront invoquées à distance. Comme ce constructeur appelle celui de la super classe **UnicastRemoteObject**, l'objet ainsi créé sera en fait transmis aux fonctions de run-time de RMI; concrètement, cela signifie que l'objet est à présent capable de recevoir des appels issus du réseau, sur un port précisé d'une manière ou l'autre. On peut encore remarquer que l'objet instancié est déclaré du type de l'interface (qui est effectivement disponible pour les clients), pas de celui de la classe qui n'est pas à la disposition du client.

Enfin, il faut donner à notre objet un nom qui permettra de le référencer. Ici entre en scène la classe **Naming**. Avec un tel nom, un disciple de CORBA s'attendrait au service de nommage. Et, de fait, son rôle est de gérer l'association entre le nom de référence et l'objet en l'enregistrant dans une "**registry**" RMI. Il s'agit en fait d'un service de nommage qui enregistre les références des objets serveurs et traite les lookups (c'est-à-dire les recherches) de ceux-ci par les clients. Il est à l'écoute sur le port standard 1099. L'enregistrement se fait en utilisant sa méthode

```
public static void bind(String name, Remote obj) throws AlreadyBoundException,  
MalformedURLException, UnknownHostException, RemoteException
```

ou même

```
public static void rebind(String name, Remote obj) throws RemoteException,  
MalformedURLException, UnknownHostException
```

pour éviter une exception dans le cas où la référence que l'on veut créer existe déjà (un objet "registrar", classiquement chargé d'enregistrer les serveurs, utilise plutôt cette dernière méthode). Le deuxième paramètre est donc l'objet Remote tandis que le premier est en fait le "*nom réseau*", donc l'URL pour le niveau applicatif auquel nous nous trouvons. Cette URL sera de la forme :

rmi://<nom de la machine hôte>:1099/RefMonitor

donc ici, par exemple :

rmi://localhost:1099/RefMonitor ou rmi://ulyssse:1099/RefMonitor.

On y distingue le protocole (rmi), le nom de la machine où le serveur est développé (localhost ou autre chose), le port (celui-ci est par défaut 1099 pour rmi) et finalement le nom proprement dit du service.

Donc, finalement :

AccesRessourceServerApplic.java

```
import java.rmi.*;  
  
import AccesRessourceServer.Interface.*;  
import AccesRessourceServer.Implementation.*;  
  
public class AccesRessourceServerApplic  
{  
    public AccesRessourceServerApplic()  
    {  
        if (System.getSecurityManager() == null)  
            System.setSecurityManager(new RMISecurityManager());  
        try  
        {  
            AccesRessourceServerInter ARSC  
            = (AccesRessourceServerInter) new AccessRessourceServerClasse();  
            Naming.rebind("rmi://ulyssse:1099/RefMonitor",ARSC);  
            System.out.println("Objet RefMonitor enregistre !!!");  
        }  
        catch (Exception e) {System.out.println("Aieeee " + e.getMessage());System.exit(0);}  
    }  
  
    public static void main (String[] arg)  
    {  
        new AccesRessourceServerApplic();  
    }  
}
```

7. Un client pour l'objet distant

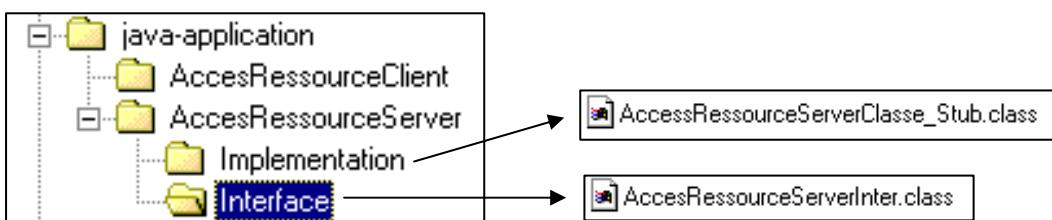
Un client très simple va se contenter de réclamer une référence à l'objet serveur distant en utilisant la méthode de la classe Naming :

```
public static Remote lookup(String name) throws NotBoundException,  
MalformedURLException, UnknownHostException, RemoteException
```

Le nom passé comme référence est le nom de service qui a été enregistré dans la registry au moyen d'un (re)bind. Bien sûr, le client devra connaître :

- ◆ l'interface de l'objet distant;
- ◆ le stub de cet objet;

Donc, sur le client :



Bien sûr, le skel ne se trouve pas sur le client ...

AccesRessourceClient.java

```
import java.rmi.*;  
import AccesRessourceServer.Interface.*;  
  
public class AccesRessourceClient  
{  
    public AccesRessourceClient () {}  
  
    public static void main(String args[])  
    {  
        if (System.getSecurityManager() == null)  
            System.setSecurityManager(new RMISecurityManager());  
        try  
        {  
            AccesRessourceServerInter referenceObjDistant =  
                (AccesRessourceServerInter)  
                Naming.lookup("rmi://ulyssse:1099/RefMonitor");  
            System.out.println("Message tiré de l'objet = " + referenceObjDistant.getMessage());  
            referenceObjDistant.setEnService("Charlet");  
            System.out.println("Message tiré de l'objet apres mise en service = " +  
                referenceObjDistant.getMessage());  
            if (referenceObjDistant.isEnService()) System.out.println("Le serveur est en service");  
            else System.out.println("RATE");  
            boolean ok = referenceObjDistant.setConnexion("Merce");  
            System.out.println("Message tiré de l'objet apres connexion Merce= " +  
                referenceObjDistant.getMessage());  
        }  
    }  
}
```

```
if (ok) System.out.println("La connexion est faite");
else System.out.println("La connexion n'est pas faite");
System.out.println("On peut utiliser " +
    referenceObjDistant.getNombreInfosDisponibles());
for (int i=0; i<4; i++)
    System.out.println("Citation = " + (String)referenceObjDistant.getNextInfo() );
}
catch (Exception e) {System.out.println("??? Aie Aie : " + e.getMessage());} }
```

8. Un exemple d'exécution du dialogue RMI

Supposons donc que nous ayons placé l'objet serveur sur la machine *ulysse* tandis qu'un objet client sera placé sur la machine *calypso*.

Le mécanisme de la registry n'est pas enclenché par défaut. On peut le démarrer sur *ulysse* par programmation ou depuis la ligne de commande en utilisant l'exécutable rmiregistry du JDK :

```
| C:\Program Files\Java\jdk1.6.0_13\bin> rmiregistry
```

Le programme reste en attente des demandes de nommage. On peut spécifier le port utilisé sur la ligne de commande si le port par défaut 1099 n'est pas utilisé. Comme ce programme n'effectue aucune sortie propre, il est sans doute plus normal de le lancer en tâche de fond (service) :

```
| C:\Program Files\Java\jdk1.6.0_13\bin> start rmiregistry
```

Nous allons ensuite instancier notre serveur rmi sur cette même machine *ulysse*. Pour cela, nous nous plaçons dans le répertoire java-application\AccesRessourceServerApplic. Il faut y lancer la commande suivant, qui appelle l'interpréteur java :

```
c:\java-application\AccesRessourceServer\Implementation>
java -Djava.rmi.server.codebase=file:/c:/java-application/ -classpath c:/java-application;
C:\Program Files\Java\jdk1.6.0_13\lib\classes.zip
AccesRessourceServerApplic.AccesRessourceServerApplic
AccesRessourceServer.Implementation.AccessRessourceServerClasse
```

On précise dans l'option codebase de java-rmi l'emplacement racine des classes du serveur, emplacement défini sous forme d'URL. De cette manière, cette information sera transmise aux clients, qui pourront ainsi charger ces classes par le biais indiqué, au moyen d'un Web serveur ou, simplement, du système de fichiers de l'hôte visé. On peut évidemment placer cette longue commande dans un fichier texte géré par le notepad ou simplement créer un fichier instrmi.bat. D'une manière ou d'une autre, on obtient :

```
| Objet RefMonitor enregistré !
```

Si, sur *calypso*, notre client a mis en place les fichiers correspondant à sa classe AccesRessourceClient ainsi que ceux qui lui sont nécessaires dans le package AccesRessourceServer (Interface et Implementation) comme expliqué au point précédent, son exécution donnera enfin :

Message tiré de l'objet = --- Access Ressource Server (c) ---
Message tiré de l'objet apres mise en service = --- Serveur mis en marche ---
Le serveur est en service
Message tiré de l'objet apres connexion Machin = --- Nouvelle connexion acceptée ---
La connexion est faite
On peut utiliser 3
Citation = Qui a coulé le Titanic ? Iceberg, encore un Juif !
Citation = Qui n'a pas connu James n'a pas connu l'Amour
Citation = Qui veut voyager loin ménage sa monture
Citation = Qui a coulé le Titanic ? Iceberg, encore un Juif !

ou si le serveur n'est pas en fonction :

Aie Aie Connection refused to host: [ulysses:1099]; nested exception is:
java.net.ConnectException: Connection refused

Remarque

On peut créer, en vue de distribution, un fichier jar contenant les différents composants utiles pour l'objet serveur distant :

```
C:\java-application\AccesRessourceServer>jar cvf AccesRessourceServer.jar *.class
adding: AccesRessourceServerInter.class (in=665) (out=337) (deflated 49%)
adding: AccessRessourceServerClasse.class (in=3006) (out=1461) (deflated 51%)
adding: AccessRessourceServerClasse_Skel.class (in=3885) (out=1917) (deflated 50 %)
adding: AccessRessourceServerClasse_Stub.class (in=5065) (out=2192) (deflated 56 %)
```

9. La nature de l'objet manipulé par le client

Pour que les choses soient parfaitement claires, on peut encore utiliser les techniques de l'introspection afin d'obtenir des renseignements sur l'objet manipulé par le client. Si on ajoute au début de ce client les lignes suivantes :

AccesRessourceClient.java (avec introspection)

```
import java.rmi.*;
import AccesRessourceServer.Interface.*;
import java.lang.reflect.*; // seul java.lang est importé d'office

public class AccesRessourceClient
{
    public AccesRessourceClient () {}

    public static void main(String args[])
    {
        if (System.getSecurityManager() == null)
            System.setSecurityManager(new RMISecurityManager());
        try
        {
            AccesRessourceServerInter referenceObjDistant =
```

```

        (AccesRessourceServerInter) Naming.lookup("rmi://claude:1099/RefMonitor");
        // à la découverte de l'objet distant
    Class info = referenceObjDistant.getClass();
    String nomC = info.getName();
    System.out.println("Nom de la classe = "+nomC);
    if (info.isInterface()) System.out.println("considéré comme interface");
    else System.out.println("considéré comme classe");
    Class inter[] = info.getInterfaces();
    for (int i=0; i<inter.length; i++)
    {
        System.out.println("interfaces["+i+"] = "+inter[i]);
    }
info=null;
try
{
    info = Class.forName
        ("AccesRessourceServer.Implementation.AccessRessourceServerClasse_Stub");
    Method infoMéthodes[] = info.getDeclaredMethods();
    for (int i=0; i<infoMéthodes.length; i++)
    {
        System.out.println("méthode["+i+"] = "+infoMéthodes[i]);
    }
}
catch (ClassNotFoundException e)
{
    System.out.println("Bof !? " + e.getMessage());
}
// suite du client
...
}
}

```

Le résultat de nos interrogations est :

Nom de la classe = AccesRessourceServer.Implementation.**AccessRessourceServerClasse_Stub**
 considéré comme **classe**
interfaces[0] = interface AccesRessourceServer.Interface.AccesRessourceServerInter
interfaces[1] = interface java.rmi.Remote
méthode[0] = public boolean
 AccesRessourceServer.Implementation.AccessRessourceServerClasse_Stub.closeConnexion
 (java.lang.String) throws java.rmi.RemoteException
méthode[1] = public java.lang.String
 AccesRessourceServer.Implementation.AccessRessourceServerClasse_Stub.getMessage()
 throws java.rmi.RemoteException
méthode[2] = public java.lang.Object
 AccesRessourceServer.Implementation.AccessRessourceServerClasse_Stub.getNextInfo()
 throws java.rmi.RemoteException
méthode[3] = public int
 AccesRessourceServer.Implementation.AccessRessourceServerClasse_Stub.getNombreClient
 sConnectés() throws java.rmi.RemoteException

```
méthode[4] = public int  
    AccesRessourceServer.Implementation.AccessRessourceServerClasse_Stub.getNombreInfos  
    Disponibles() throws java.rmi.RemoteException  
méthode[5] = public boolean  
    AccesRessourceServer.Implementation.AccessRessourceServerClasse_Stub.isEnService()  
    throws java.rmi.RemoteException  
méthode[6] = public void  
    AccesRessourceServer.Implementation.AccessRessourceServerClasse_Stub.setALARret(java.  
    lang.String) throws java.rmi.RemoteException  
méthode[7] = public boolean  
    AccesRessourceServer.Implementation.AccessRessourceServerClasse_Stub.setConnexion(ja  
    va.lang.String) throws java.rmi.RemoteException  
méthode[8] = public void  
    AccesRessourceServer.Implementation.AccessRessourceServerClasse_Stub.setEnService(jav  
    a.lang.String) throws java.rmi.RemoteException
```

Difficile de terminer cette introduction aux objets distribués sans évoquer la couche supérieure à RMI : les EJBs ...

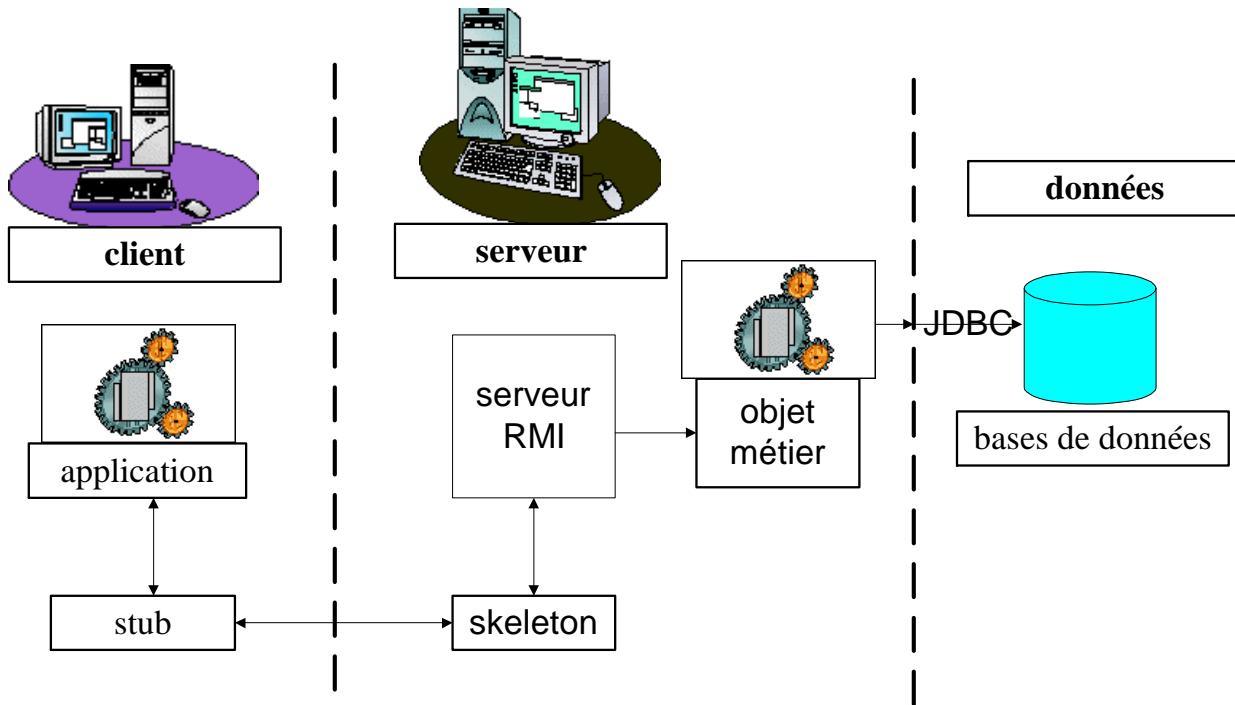
10. Les objets distants : la suite avec les EJB

10.1 L'architecture 3-tiers

L'architecture 3-tiers classique peut parfaitement utiliser la "**boucle RMI**" décrite ci-dessus. En effet,

- ◆ le client invoque une méthode d'un objet distant au moyen du stub; celui-ci possède les mêmes méthodes que cet objet distant, mais son implémentation comporte en fait seulement les opérations réseaux vers le serveur;
- ◆ le serveur reçoit l'invocation par son skeleton et va invoquer la méthode demandée; cette fois, l'implémentation de cette méthode comporte effectivement ce que l'on appelle la "logique métier", c'est-à-dire le traitement demandé;
- ◆ le SGBD (par exemple) qui constitue le 3ème tiers sera alors mis à contribution pour l'accès aux données.

Schématiquement :



C'est dans ce contexte 3-tiers qu'intervient la spécification des **EJB** (Enterprise Java Beans), qui en est actuellement à sa version 3.0, après une version 2.0 encore bien implantée.

10.2 Les Enterprise Java Beans

Les **EJB** (Enterprise Java Beans) sont des composants :

- ◆ qui peuvent s'**assembler** pour constituer une application serveur accessible dans un contexte d'**objets distribués** (de type RMI);
- ◆ dont le but est **centraliser la logique** de cette application (on parle encore d'e"logique métier"), laissant les questions de **présentation** et d'accès à d'autres composants comme les servlets et les Java Server Pages;
- ◆ qui sont instanciés au sein d'un **container** qui les enregistre, les rends accessibles et gère pour eux les questions de sécurité et de transactions;
- ◆ qui sont portables sur toute plate-forme **J2EE**;
- ◆ qui sont **paramétrables**.

Contrairement à ce que l'on pourrait croire d'après leur nom, les **EJB** ne sont pas une extension des Java Beans (évoqués dans un chapitre précédent). Cependant, dans les deux cas, il s'agit d'un modèle de **composants**, développés pour une certaine action et pas pour une application bien déterminée; tous deux possèdent des capacités d'introspection et tous deux utilisent les mêmes conventions concernant les méthodes `getxxx()` et `setxxx()` concernant des propriétés. Mais

les **Java Beans** sont des composants destinés à être utilisés **au sein d'une application cliente** tandis que les **Enterprise Java Beans** sont destinés à être utilisés **du côté serveur**, dans un contexte distribué, c'est-à-dire entre applications.

En ce sens, les EJB rappellent les objets **RMI**. Mais ils renferment également la notion de **transaction**. Les EJB utilisent en effet les **Java Transaction API (JTA)** pour gérer les transactions. Dans ce contexte, on utilise à présent l'acronyme de **CTM (Component Transaction Monitor)** pour désigner des serveurs d'objets distribués qui maintiennent les transactions. En ce sens, ils sont les enfants naturels des moniteurs transactionnels (**Transaction Processing monitor – TP**), qui gèrent l'ensemble de l'environnement d'exécution d'une application, transactions comprises, et des bus à objets (**Object Request Broker – ORB**). Le package `javax.transaction` fournit effectivement une classe `UserTransaction`, dont les méthodes `commit()` et `rollback()` se passent de commentaires. Une utilisation classique est alors :

```
UserTransaction tr = ... // obtenir l'objet de manière appropriée  
tr.begin();  
try  
{    ... // opérations de la transaction  
    tr.commit(); // tout s'est bien passé : on confirme  
}  
catch (Exception e)  
{  
    tr.rollback(); // si cela tourne mal, on oublie  
}
```

Les APIs des EJB proprement dits se trouvent dans le package **javax.ejb**, disponible sur le site de Sun. Mais ce package est essentiellement fourni pour assurer que les composants distribués vont interagir selon une spécification communément admise. Les interfaces sont ceux spécifiés par Java RMI-IIOP : les types utilisés sont les types primitifs et les classes qui implémentent l'interface `Serializable`.

De plus, il faut impérativement disposer d'un serveur EJB. A l'image de l'architecture RMI, un objet "client EJB" s'adresse à un objet distant (appelé "objet EJB" dans la suite) qui agit comme une sorte de serveur. On retrouve donc les intervenants de RMI. Mais l'architecture EJB prévoit un troisième acteur appelé le **container**. Ce container, objet se trouvant du côté serveur exclusivement, a pour rôle de gérer les objets distants situés sur le serveur en les enregistrant, en leur permettant leur "exportation" et en leur fournissant toute une série de services comme la gestion des transactions, celle des ressources, la sécurité, la persistance, ... Plus concrètement, il est responsable de la génération des diverses classes qui vont permettre la connexion des composants, en particulier des stubs que le client EJB utilisera.

Les EJBs méritent à eux seuls tout un cours (tâche que nous laisserons à d'autres ;-)¹⁶...). Nous allons donc nous contenter ici de donner une petite idée de leur utilisation.

¹⁶ voir les cours de T.Bastianelli (le fondateur) et M.Madani (le maître actuel) à ce sujet ...

10.3 **Deux types de beans**

L'architecture des EJB distingue deux types de beans :

- ◆ les beans "entité" [**entity beans**] : en première approximation, il s'agit d'informations contenues dans une base de données (par exemple, un tuple, un record, etc), donc une espèce de vue au sens des SGBD relationnels; la notion de persistiance est donc bien présente; de plus, un tel bean fournit une logique indépendamment du client – autrement dit, il n'est pas lié à la durée de vie des sessions des clients;
ex: un bean qui offre aux clients la possibilité de rechercher un article de stock au moyen de son libellé (qui sert de clé primaire) pour consulter ou modifier son prix.
- ◆ les beans "session" [**session beans**] : toujours en première approximation, il s'agit d'une extension de la notion d'application, puisqu'un session bean fait quelque chose, avec souvent un résultat sur l'un ou l'autre entity bean; il n'y a pas de notion de persistiance; cette fois, comme leur nom l'indique d'ailleurs, ils sont dédicacés à un ou plusieurs clients bien précis, pouvant notamment conserver l'état d'un seul client c'est-à-dire des informations sur celui-ci (le bean est alors dit **Stateful**) ou pas (le bean est dit **Stateless** – un tel bean est donc surtout utilisé pour des tâches génériques).

11. **Les éléments programmatiques d'un EJB : interfaces et classes (2.0)**

Afin de conclure sur une note plus concrète, développons quelque peu les mécanismes à mettre en place pour construire un bean entité (par exemple). Son rôle est de manipuler le prix d'articles connus par leur libellé; du point de vue base de données, ce libellé est la clé primaire de la table "articles" que l'on devine en filigranes.

Concrètement, l'implémentation d'un objet EJB réclame plusieurs éléments que nous allons développer ici. Les packages javax.ejb et javax.ejb.deployment seront nécessaires, mais aussi ceux de **JNDI** (Java Naming and Directory Interface – il s'agit d'un ensemble d'APIs permettant de retrouver des ressources distantes sur le réseau – service de nommage ?), soit javax.naming, javax.naming.directory et javax.naming.spi.

De quoi a-t-on besoin pour utiliser un EJB ? Bien logiquement d'un objet dont la classe comporte les méthodes utilisables par le client et les méthodes de gestion de l'EJB (création destruction, localisation). Ceci correspond à deux interfaces.

11.1 **L'interface distante du composant**

Il s'agit de l'interface qui définit les méthodes du composant EJB qui seront invoquées à distance par les objets clients; on parle encore, plus généralement, de *Component interface*. Cet interface dérive toujours de l'interface **EJBObject**, qui lui-même, sans surprise, dérive de l'interface **Remote** évoqué dans RMI; l'exception **RemoteException** (déjà connue depuis RMI) est évidemment lancée dès qu'un problème réseau se fait jour. Cet interface **EJBObject** ne comporte que quelques méthodes, comme :

```
public EJBHome getEJBHome() throws java.rmi.RemoteException  
public void remove() throws java.rmi.RemoteException, RemoveException  
ou  
public java.lang.Object getPrimaryKey() throws java.rmi.RemoteException  
cette dernière n'ayant évidemment de sens que pour un bean entité.
```

Pour l'interface de notre composant, cela pourrait donner :

ArticleRemoteInter.java

```
import java.rmi.*;
import javax.ejb.*;

public interface ArticleRemoteInter extends EJBObject
{
    public String getLibelle() throws RemoteException
    public float getPrix() throws RemoteException
    public void setPrix(float p) throws RemoteException
}
```

Les clients se serviront de l'implémentation de cet interface par un stub pour utiliser les méthodes d'un objet EJB précis.

Remarque

En fait, depuis la spécification 2.0, l'EJB et son client peuvent s'exécuter aussi sur la même JVM. Dans ce cas, l'interface du composant dérive alors de l'interface **EJBLocalObject**, défini dans le même package. On ne parle plus alors que d'"**interface du composant**" au lieu d'"interface distante".

11.2 L'interface locale de gestion de l'EJB

Il doit définir le "cycle de vie" de l'objet bean, c'est-à-dire plus concrètement définir les méthodes de création, de recherche et de destruction de l'objet EJB – il s'agit donc clairement de l'interface d'un "bean factory" au sens des méthodes factory évoquées en d'autres lieux; on parle encore, plus généralement, de *Home interface* (en français, "interface d'accueil"). Cette fois, cet interface n'a pas d'équivalent RMI; il dérive toujours de l'interface **EJBHome**, qui lui-même dérive de l'interface **Remote**. **Les clients se serviront de l'implémentation de cet interface pour créer ou pour gérer un objet EJB précis.** A nouveau, on ne trouve dans cet interface que quelques méthodes :

```
public EJBMetaData getEJBMetaData() throws java.rmi.RemoteException
    où l'interface EJBMetaData contient des méthodes prévisibles comme
        public java.lang.Class getRemoteInterfaceClass()
        public java.lang.Class getRemoteInterfaceClass()
```

```
public void remove(java.lang.Object primaryKey) throws java.rmi.RemoteException,
    RemoveException
```

Vu le rôle dévolu aux classes qui implémenteront cet interface (c'est-à-dire à gérer l'existence du bean), on conçoit aisément qu'il doit aussi comporter une ou plusieurs méthodes nommées **create()** (à chacune de ces méthodes correspondra une méthode ejbCreate() dans la classe de l'EJB) et **remove()**. De plus, dans le cas d'un EJB entity, il faut prévoir le moyen de rechercher un tel objet selon l'un ou l'autre critère – il faut donc prévoir des méthodes de type **findXXX()**, comme par exemple findByPrimaryKey(). Pour notre exemple :

ArticleHomeInter.java

```
import java.rmi.*;
import javax.ejb;
```

```
public interface ArticleHomeInter extends EJBHome
{
    public ArticleRemoteInter create (String l, float p)
        throws CreateException, RemoteException
    public ArticleRemoteInter findByPrimaryKey (String l)
        throws FinderException, RemoteException
}
```

Remarque

Dans le cas où l'EJB et son client s'exécutent aussi sur la même JVM, l'interface à utiliser dérive alors de l'interface **EJBLocalHome**, défini dans le même package.

11.3 Une classe bean

Elle implémente effectivement, du côté serveur, la logique de l'objet EJB en définissant ses méthodes – l'analogie avec RMI se confirme. Parmi ces méthodes, on devra en trouver qui

- ◆ implémentent celles de l'interface remote (sans pour autant que le bean implémente cet interface);
- ◆ correspondent à celles de l'interface home (par exemple, si cet interface home comporte une méthode **create()**, le bean comportera une méthode **ejbCreate()** avec les mêmes paramètres);
- ◆ implémentent les méthodes invoquées par le container lorsqu'il manipule les beans; sans surprise, on les appelle des méthodes de callback puisqu'elles permettent de notifier au composant les événements qui affectent son cycle de vie. Ces méthodes sont déclarées dans des interfaces, respectivement **EntityBean** ou **SessionBean**; pour notre exemple, c'est le premier. On y trouve, par exemple :

```
public void ejbRemove() throws RemoveException, EJBException,
    java.rmi.RemoteException
public void ejbActivate() throws EJBException, java.rmi.RemoteException
public void ejbPassivate() throws EJBException, java.rmi.RemoteException
public void ejbLoad() throws EJBException, java.rmi.RemoteException
public void ejbStore() throws EJBException, java.rmi.RemoteException
```

Ici :

ArticleBean.java

```
import javax.ejb.*;
public class ArticleBean implements EntityBean
{
    private String libelle;
    private float prix;

    // Méthode de création du bean : elle est appelée avant l'insertion du tuple
    // correspondant dans la base de données
    public void ejbCreate (String l, float p) throws CreateException
    {
        libelle=l; prix=p;
    }
}
```

```
// Initialisations complémentaires
public void ejbPostCreate (String l, float p) throws CreateException, RemoteException { }

// Méthodes offertes par le bean; ce sont celles de l'interface remote
public String getLibelle() { return libelle; }
public float getPrix() { return prix; }
public void setPrix(float p) { prix = p; }

// Méthodes callback – déclarées dans EntityBean
public void ejbActivate() throws RemoteException {}
public void ejbLoad() throws RemoteException {}

...
)
```

Mais, pour un EJB entité, il manque quelques chose ...

11.4 Une classe clé primaire

On s'en doute donc, elle n'est nécessaire que pour un bean entité. Son grand mérite est d'encapsuler une clé primaire au sens SGBD, dissimulant éventuellement le fait que cette clé est composite. Il lui est demandé d'implémenter Serializable.

ArticlePrimaryKey.java

```
import java.io.*;

public class ArticlePrimaryKey implements Serializable
{
    private String pk;

    public getPK() {return pk; }
}
```

Notre bean peut alors être complété des méthodes ejbFindXXX qui correspondent aux méthodes findXXX de l'interface de gestion du bean :

ArticleBean.java (2)

```
import javax.ejb.*;

public class ArticleBean implements EntityBean
{
    private String libelle;
    private float prix;

    public void ejbCreate (String l, float p) throws CreateException
    { libelle=l; prix=p; }

    public ArticlePrimaryKey findByPrimaryKey (String l) throws FinderException { ... }

    ...
    public void ejbActivate() throws RemoteException { ... }
    public void ejbLoad() throws RemoteException { ... }
    ...
)
```

11.5 Le container d'EJB

C'est le rôle du **container** d'enregistrer les interfaces home de chaque bean sous un nom appartenant à un espace de nom JNDI; les clients pourront de cette manière trouver ces interfaces et les utiliser pour créer les beans. Le container est responsable de la génération des stubs et des objets du côté du serveur qui implémentent les interfaces remote et home. Quand un client cherchera l'interface home d'un bean au moyen de JNDI, il recevra en réponse une instance du stub home. Via RMI, les méthodes de ce stub seront invoquées à distance sur l'objet home du côté du serveur.

Le client EJB n'interagit jamais directement avec l'objet EJB : il utilise, pour obtenir les services de celui-ci, les méthodes des deux interfaces home et remote, fournies par les stubs que le container aura générés. Ce container va également construire une implémentation de ces mêmes interfaces mais du côté serveur. Le dialogue client-serveur se fera donc par l'intermédiaire de ces deux paires d'implémentation des interfaces remote et home.

11.6 Le descripteur de déploiement

La manière dont le bean sera effectivement géré par le container est précisé au moyen d'un **descripteur de déploiement**. Celui-ci joue un rôle analogue à celui d'un fichier web.wml ou properties pour les applications WEB : il décrit les interfaces et les classes impliquées, l'utilisation des transactions et la gestion du bean au moment de l'exécution. Dans la version 1.0 des EJB, le descripteur de déploiement était une classe sérialisable. Il s'est transformé en un fichier XML (Extensible Markup Language) pour la version 1.1. En général, ce fichier XML est généré par l'IDE utilisé ☺, mais la version standard comporte des informations du type :

```
<ejb-jar xmlns:http://java.sun.com/xml/ns/j2ee ...>

<enterprise-beans>
  <entity>
    <display-name>ArticleBean</display-name>
    <ejb-name>ArticleBean</ejb-name>
    <home>ArticleHomeInter</home>
    <remote>ArticleRemoreInter</remote>
    <ejb-class>ArticleBean</ejb-class>
    ...
  </entity>
</enterprise-beans>
</ejb-jar>
```

11.7 Le client

Le client qui souhaite utiliser un objet article le fera en utilisant des APIs **JNDI**. Qu'il nous suffise de savoir ici qu'il faut créer un contexte, que l'on peut assimiler à une espèce de point de départ pour la recherche de notre objet article selon son nom (ici, ArticleRemoteInter) qu'il permettra. La méthode lookup() de la classe Context permet de retrouver l'objet cherché. Mais il faut encore caster le résultat en ArticleHomeInter, puisque celui-ci qualifie en fait le stub utilisable; la méthode **narrow()**, de javax.rmi.PortableRemoteObject, réalise cette transformation en assurant la compatibilité avec les serveurs RMI-IIOP. Ceci se programme simplement par :

```
import javax.naming.*;
import javax.rmi.PortableRemoteObject;

Context c = new InitialContext();
Object o = c.lookup("ArticleRemoteInter");
ArticleHomeInter a =
    (ArticleHomeInter)PortableRemoteObject.narrow(o, ArticleHomeInter.class)
```

Ajouter un tuple à la base de données se demandera simplement par :

```
ArticleRemoteInter ad = a.create("céréales", 78);
tandis que rechercher un article se demandera par :
```

```
ad = a.findByPrimaryKey ("preservatif");
System.out.println("prix = " + ad.getPrix().toString());
```

12. Les éléments programmatiques d'un EJB : interfaces et classes (3.0)

La norme 3.0 des EJB a considérablement simplifié le développement, d'autant qu'un outil comme Netbeans 6.* fournit des outils simples et puissants. A titre documentaire (encore une fois, le cours de M. Madani couvre le sujet), on pourra comparer le code obtenu pour un projet Enterprise Calculateur dans la nouvelle norme : les annotations simplifient tout ! Déployé sur le serveur Glassfish, cela donne simplement :

12.1 L'interface distante du composant

CalculateurRemote.java

```
package calcul;
import javax.ejb.Remote;
@Remote
public interface CalculateurRemote
{
    int add(int a, int b);
    String hello(String nom);
}
```

12.2 La classe bean

CalculateurBean.java

```
package calcul;
import javax.annotation.security.*;
import javax.ejb.Stateless;
@DeclareRoles("admin")
@Stateless
public class CalculateurBean implements CalculateurRemote
{
    public int add(int a, int b) { return a+b; }
    @RolesAllowed("admin")
    public String hello(String nom) { return "Hello " + nom; }
}
```

12.3 Le client

Main.java

```
package login;

import calcul.CalculateurRemote;
import javax.ejb.EJB;
import javax.swing.*;

public class Main
{
    @EJB
    private static CalculateurRemote calculateurBean;

    public static void main(String[] args)
    {
        JOptionPane.showMessageDialog(null, "2 + 2 = " + calculateurBean.add(2, 2));
        JOptionPane.showMessageDialog(null, calculateurBean.hello("Mr admin"));
    }
}
```

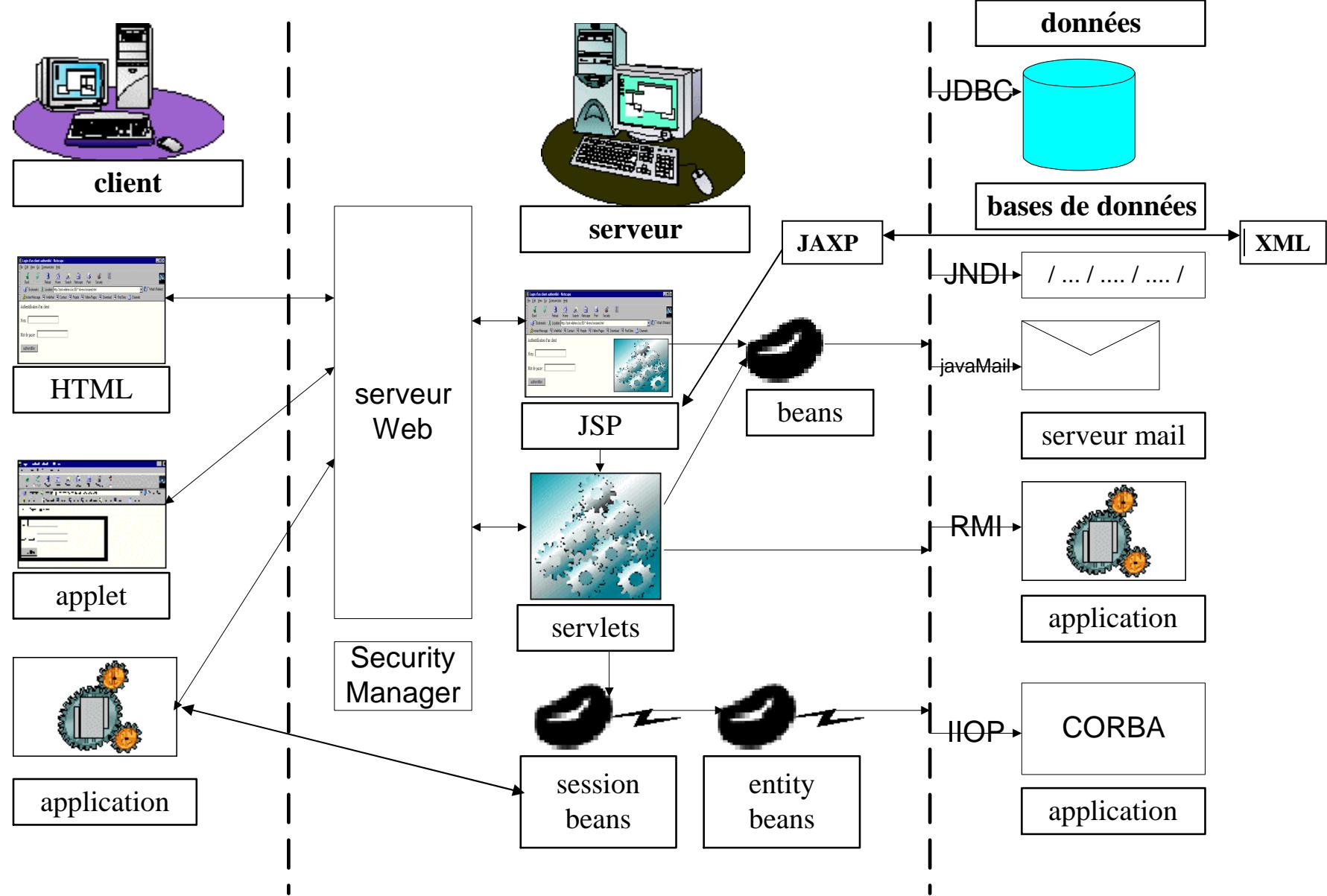
13. La plate-forme de développement J2EE

13.1 Le panorama général de J2EE

On l'a déjà dit, les développements les plus récents en Java, pour ce qui concerne les serveurs, s'effectuent sur base de la plate-forme générale **J2EE** (**J**ava **2** **E**nterprise **E**dition), une spécification développée par Sun en collaboration avec d'autres vendeurs de technologies Internet et middlewares. Pour rappel, elle définit

- ◆ des architectures de programmation (orientées WEB et orientées Objets distribués);
- ◆ des APIs fournissant divers services au moyen de techniques qui constituent des extensions par rapport au Java de base : JDBC, RMI, JNDI, EJB, Java IDL, les servlets et JSP, JAXP, etc;
- ◆ un environnement d'exécution (runtime) permettant aux applications d'accéder aux implémentations de ces APIs.

Le schéma suivant donne une idée de cette plate-forme. On y remarquera que, typiquement, les beans session sont invoqués par le client et utilisent les services des beans entité qui, eux, ne sont jamais passés à ce client :



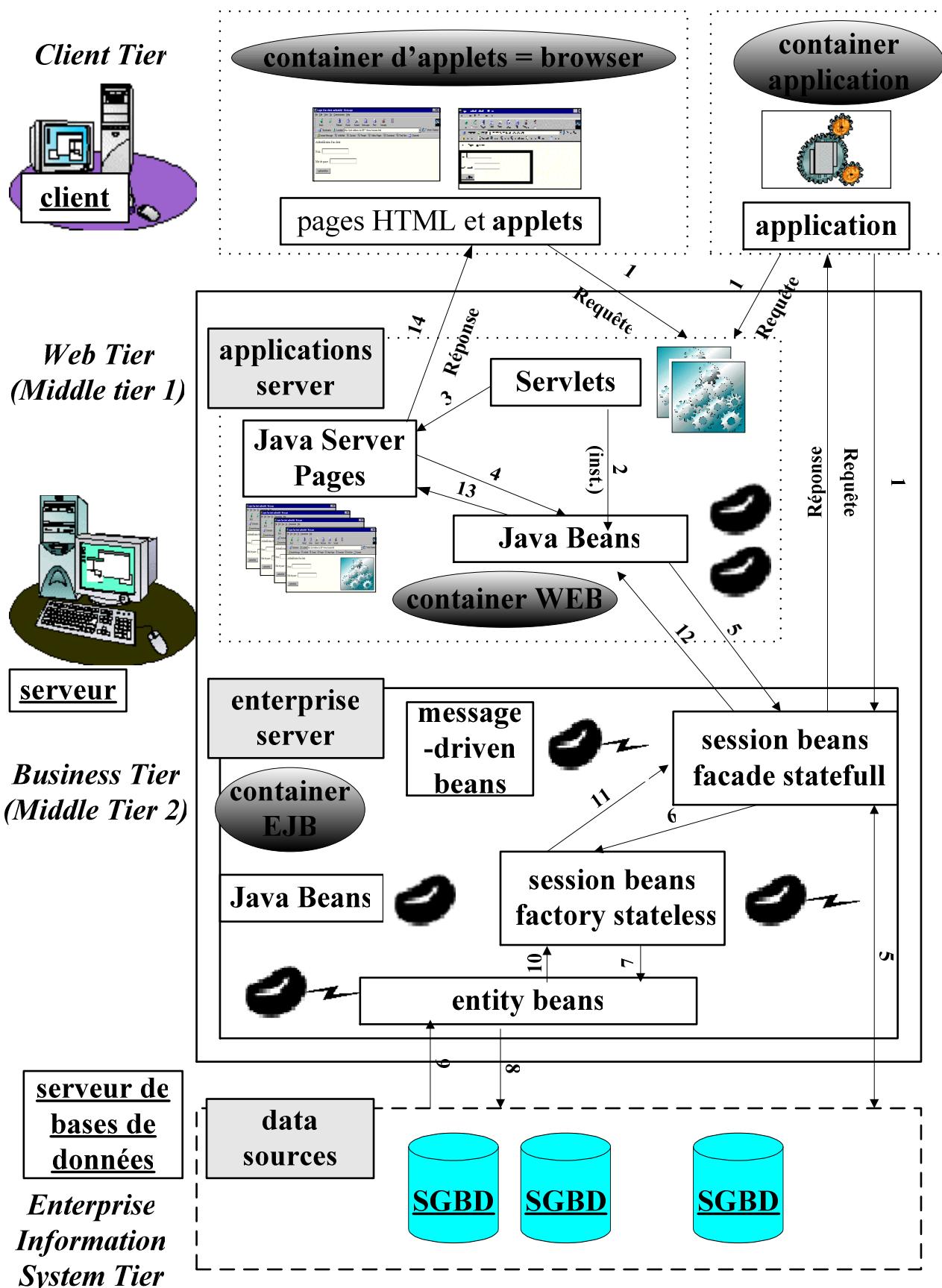
13.2 Le modèle multi-tiers de J2EE

Le schéma général évoqué ci-dessus peut être défini plus précisément en utilisant les termes appropriés de la spécification de la plate-forme J2EE. Celle-ci distingue divers types de **composants** [*components*], qui sont en fait *des classes Java mais écrites de manière telle à pouvoir interagir les unes avec les autres selon les spécifications de la plate-forme* (on peut considérer qu'il s'agit ici d'une généralisation de la philosophie de conception des Java Beans). Ces types de composants sont :

type	machine hôte	exemples
composants client	machine client	pages HTML + browser, applets + JVM
composants Web	machine serveur	servlets et JSPs + Java Beans*
composants business	machine serveur	EJBs + Java Beans*
composants Enterprise Information System	machine SGBD	procédures stockées + Java Beans*

(* au sens strict, les Java Beans ne sont pas considérés comme des composants J2EE et disposent de leur propre spécification)

Le schéma général est alors celui-ci :



Classiquement, on y distingue :

a) **les composants clients**

i) le "*client léger*" [*thin client*] qui comporte

- des pages HTML statiques et/ou dynamiques (docn générées par les composants Web);
 - des applets Java;
 - un browser avec les plug-ins nécessaires (une JVM pour les applets par exemple).
- ii) l'*application cliente*, qui travaille avec
- la ligne de commande (très en vogue à Sparte ;-));
 - un GUI construit avec AWT ou Swing.

Une telle application s'adresse le plus souvent directement aux EJBs de la couche business, sans passer donc par une couche Web. Cependant, rien ne lui interdit de créer une connexion HTTP vers cette couche – c'est l'idée de la technologie **Java WebStart**.

b) **les composants WEB**

Ils nous sont évidemment bien connus, puisqu'il s'agit des *servlets* et des *Java Server Pages*. Ils peuvent faire usage

- de Java Beans;
- de pages statiques HTML;
- de classes diverses constituant les APIs de diverses technologies, comme les Java Mail ou JDBC.

Tous ces éléments additionnels ne sont pas considérés comme des composants J2EE.

c) **les composants business**

Il s'agit ici des *Enterprise Java Beans*, avec leurs trois types :

- beans session pour la logique métier;
- beans entités pour les données persistantes;
- beans orientés messages pour les beans sessions listener JMS.

d) **les composants EIS**

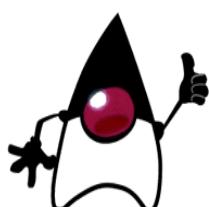
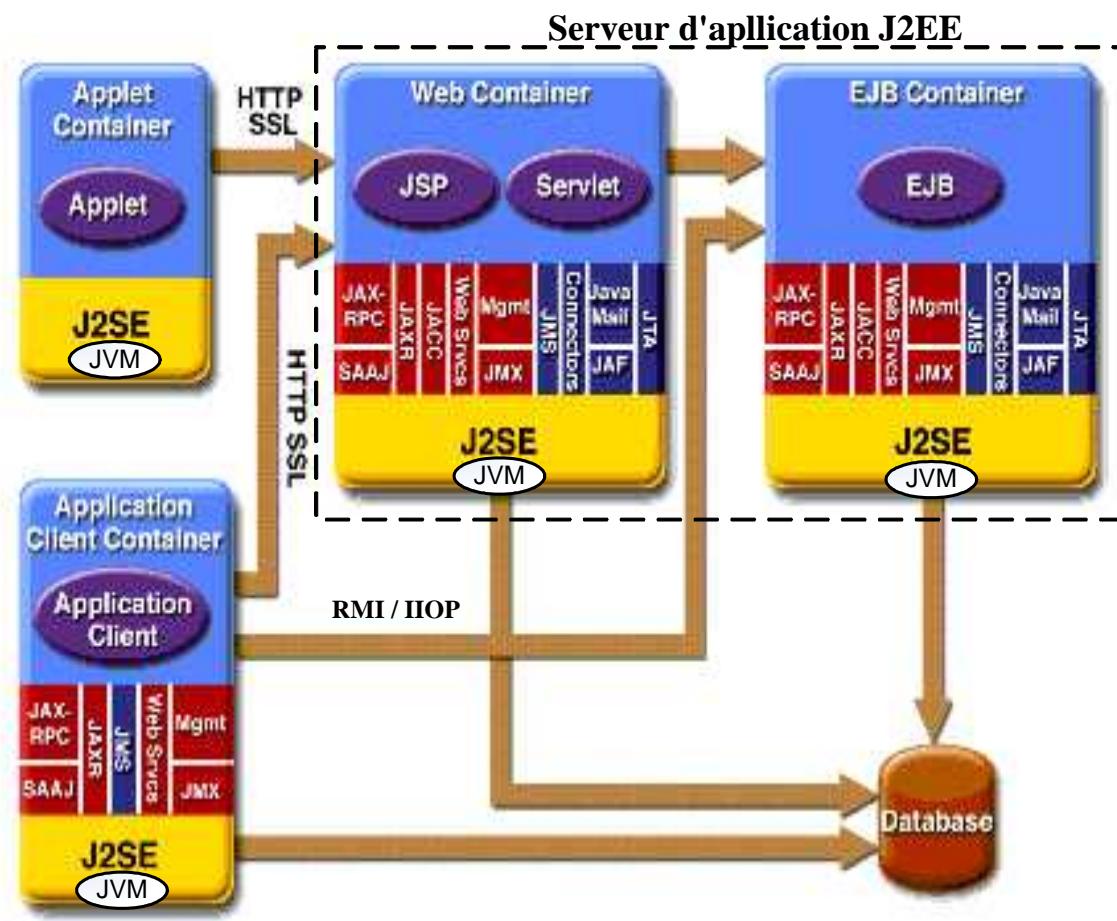
Ces composants ne relèvent pas forcément de Java : ce sont tous les outils des systèmes d'informations comme les procédure stockées des SGBDs.

Ces divers composants doivent être instanciés au sein d'un environnement qui va leur permettre de fonctionner, notamment en y incorporant les APIs relevant de diverses technologies (JDBC, JAXP, Java Mail, etc). Comme nous le savons, un tel environnement est encore appelé un **container**. Dans le modèle J2EE, on peut distinguer

- côté serveur, le Web container et l'EJB container;
- côté client, l'Applet container et l'Application client container.

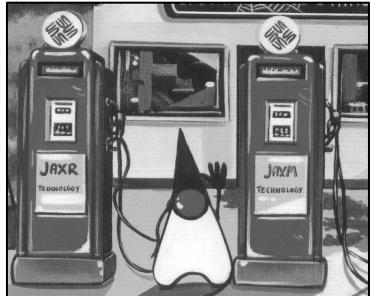
Un serveur J2EE a pour mission de fournir les containers Web et Enterprise.

Schématiquement et finalement :



La grande mode de l'informatique distribuée dans le monde Windows est à présent celle des services Web. Java ne pouvait y être insensible ;-) – une petite introduction ?

XX. Une introduction aux services Web en Java



On n'est jamais si bien servi que par soi-même.

(Charles Guillaume Etienne, Brueys et Palaprat)

1. Un contexte d'informatique distribuée

Bien qu'une définition générale et acceptée par tous les développeurs soit probablement impossible à fournir, disons que l'on entend par "service Web" [*Web service*] toute unité programmatique ou tout système logiciel

- ◆ jouant le rôle de serveur, c'est-à-dire assurant un **service**;
- ◆ qui a **publié** la manière de l'utiliser, autrement dit dont l'interface peut être obtenu;
- ◆ qui peut être accédée par un client de **nature quelconque**, pour peu que celui-ci respecte les règles d'utilisation.

Par rapport aux EJB déjà évoqués précédemment, la grande différence est que *le client n'est pas forcément une entité Java* (plus généralement, une entité programmée selon le même plate-forme que le client). La requête doit donc être exprimée de manière telle qu'elle soit indépendante de la technologie utilisée par le client comme de celle utilisée par le service : c'est le rôle du protocole SOAP de gérer cette contrainte en fournissant des données auto-descriptives.

Quoi qu'il en soit, nous nous trouvons clairement, à nouveau, dans le contexte des systèmes distribués. Nous allons donc ici nous contenter d'une petite introduction, afin de brosser un éventail plus ou moins complet des développements Web en Java, mais nous laisserons à d'autres le soin d'une étude exhaustive¹⁷.

2. Un exemple : le service des provinces

Afin de fixer les idées, considérons le service Web élémentaire qui fournit comme réponse le nom du chef lieu de la province figurant dans la requête qu'il reçoit; il est aussi capable de fournir la liste complète province – chef lieu. Du point de vue d'un programmeur Java, ce "service" n'est rien d'autre qu'une banale classe qui recherche le chef-lieu dans un objet Properties préalablement chargé à partir d'un fichier :

ServiceProvince.java

```
/*
 * ServiceProvince.java
 * Created on 26 avril 2004, 17:26
 */
```

¹⁷ voir à ce sujet les cours de M.Madani et T.Bastianelli "Systèmes distribués : Web services" (assez orienté .NET) et le TFE de S.Reip "Etude de diverses implémentations des services Web sous Java" (très orienté Java).

```

/**
 * @author Vilvens
 */
import java.util.*;
import java.io.*;

public class ServiceProvince
{
    static private String FICHIER_PROVINCES_CHEFS_LIEUX =
        "c:\\java-sun-application\\WebServiceProvince\\provinces-chefs-lieu.properties";
    Properties pCl;

    public ServiceProvince()
    {
        pCl = new Properties();
        try
        {
            pCl.load(new FileInputStream(FICHIER_PROVINCES_CHEFS_LIEUX));
        }
        catch (IOException e)
        {
            System.out.println("Erreur de lecture du fichier properties : " + e.getMessage());
        }
    }

    public String getChefLieu (String n)
    {
        String res = pCl.getProperty(n);
        if (pCl==null) return "Province inconnue";
        else return res;
    }

    public void afficheListe (PrintStream s) { pCl.list(s); }
}

```

Comment à présent déployer ce service pour qu'il soit accessible à tous ? Sans doute faut-il imaginer un moyen pour que tout client intéressé puisse faire comprendre au service qu'il souhaite, par exemple, le chef-lieu d'une province donnée ...

3. Le protocole SOAP

SOAP (Simple Object Access Protocol) est un protocole basé sur XML dont l'objectif est de faciliter l'échange d'informations dans un milieu distribué et hétérogène. Soutenu par le W3C, Sun, Oracle, Microsoft, IBM, etc, il est en fait LE protocole standard des services Web.

Clairement "orienté transmission", il traite exclusivement du contenu et du format des données transmises entre les intervenants en étant auto-descriptif. Il n'est donc pas "orienté API" puisqu'il ne fournit pas les commandes permettant d'accéder à ce contenu. Les messages SOAP peuvent être véhiculés dans la trame de divers protocoles (SMTP par exemple), bien que l'utilisation courante soit d'utiliser HTTP.

En gros, on peut dire que le protocole SOAP fait intervenir trois composantes :

- l'enveloppe qui décrit le contenu d'un message et donne quelques indications sur la manière de le traiter;
- un ensemble de règles décrivant l'encodage des instances des types de données qui vont être manipulées;
- une description de l'utilisation de l'enveloppe et des règles d'encodage de données permettant de rédiger une requête ou une réponse.

Un message SOAP très simple peut par exemple ressembler à ceci (en l'occurrence, il s'agit d'une requête pour notre service de chef-lieu de provinces) :

Un exemple de requête SOAP

```
<?xml version="1.0" encoding="UTF-8"?>

<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <soapenv:Body>
        <getChefLieu soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
            <arg0 xsi:type="xsd:string">Hainaut</arg0>
        </getChefLieu>
    </soapenv:Body>
</soapenv:Envelope>
```

On désigne sous le nom d"**enveloppe**" l'intégralité d'une requête ou d'un message SOAP. On y place donc tout ce qui peut être utile, le destinataire étant sensé savoir interpréter ce contenu XML en le parsant. Le terme "enveloppe" constitue donc la balise de plus haut niveau du document XML qu'est le message SOAP. En réalité, les balises SOAP sont définies dans un espace de nom <http://schemas.xmlsoap.org/soap/envelope> - l'identifiant en est "soapenv" dans l'exemple ci-dessus. Schématiquement, un message SOAP obéit à la structure suivante :

<soapenv:Envelope	tag racine obligatoire de tout document SOAP valide
xmlns:soapenv	espace de nom obligatoire pour désigner l'environnement
xxxx:yyyyy ... >	d'autres espaces de nom qui seront utilisés pour interpréter le message (par exemple, le type d'un paramètre)
<soapenv:Header> </soapenv:Header>	tag en-tête facultatif : comporte des informations d'authentification, de paramétrage de traitement, etc
<soapenv:Body>	tag obligatoire qui marque le début du message proprement dit (par exemple, pour une requête : que veut-on ? en fonction de quoi ?)
<nom_methode> <arg0>... </nom_methode>	typiquement, le nom de la méthode à invoquer et le(s) paramètre(s)

<soapenv:Fault> <faultcode>...</faultcode> <faultstring>... </faultstring> ... </soapenv:Fault>	tag facultatif pour expliquer une erreur à l'initiateur d'un message
</soapenv:Body>	
</soapenv:Envelope>	

Au-delà de ces règles abstraites vaguement décrites ci-dessus, il existe des dizaines d'implémentations concrètes de SOAP, certaines en Java (Apache Soap, Apache Axis, d'autres pas (.NET, PHP)). Celle d'Apache est l'une des plus connue, mais ceci ne résout rien : l'interopérabilité visée est parfois délicate à obtenir. En fait, le client potentiel ne connaît pas forcément la manière de s'adresser au service ni le type des paramètres de la requête ou celui de la réponse. Que faire ?

4. Le langage WSDL

Afin de surmonter les problèmes d'interopérabilité dus aux diverses implémentations de SOAP, on a imaginé d'utiliser un langage de description des fonctionnalités du service qui soit standard pour tous. Le langage **WSDL** (Web Service Description Language) permet de rédiger un document XML qui décrit tout ce qui peut être utile à un client du service. L'idée est qu'il suffit alors d'interroger la plate-forme de déploiement des services Web pour obtenir les caractéristiques de ceux-ci.

La description d'un service peut, par exemple (toujours notre exemple de province), ressembler à ceci :

Un exemple de document WSDL

```
<?xml version="1.0" encoding="UTF-8" ?>
<wsdl:definitions targetNamespace="http://myriam:8080/axis/ServiceProvince.jws"
xmlns="http://schemas.xmlsoap.org/wsdl/" xmlns:apachesoap="http://xml.apache.org/xml-soap"
xmlns:impl="http://myriam:8080/axis/ServiceProvince.jws"
xmlns:intf="http://myriam:8080/axis/ServiceProvince.jws"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <wsdl:message name="getChefLieuRequest">
        <wsdl:part name="n" type="xsd:string" />
    </wsdl:message>
    <wsdl:message name="afficheListeResponse" />
    <wsdl:message name="getChefLieuResponse">
        <wsdl:part name="getChefLieuReturn" type="xsd:string" />
    </wsdl:message>
    <wsdl:message name="afficheListeRequest">
        <wsdl:part name="s" type="xsd:anyType" />
    </wsdl:message>
```

```

<wsdl:portType name="ServiceProvince">
    <wsdl:operation name="afficheListe" parameterOrder="s">
        <wsdl:input message="impl:afficheListeRequest"
                    name="afficheListeRequest" />
        <wsdl:output message="impl:afficheListeResponse"
                    name="afficheListeResponse" />
    </wsdl:operation>
    <wsdl:operation name="getChefLieu" parameterOrder="n">
        <wsdl:input message="impl:getChefLieuRequest"
                    name="getChefLieuRequest" />
        <wsdl:output message="impl:getChefLieuResponse"
                    name="getChefLieuResponse" />
    </wsdl:operation>
</wsdl:portType>

<wsdl:binding name="ServiceProvinceSoapBinding"
              type="impl:ServiceProvince">
    <wsdlsoap:binding style="rpc"
                      transport="http://schemas.xmlsoap.org/soap/http" />
    <wsdl:operation name="afficheListe">
        <wsdlsoap:operation soapAction="" />
        <wsdl:input name="afficheListeRequest">
            <wsdlsoap:body
                encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                namespace="http://DefaultNamespace" use="encoded" />
            </wsdl:input>
        <wsdl:output name="afficheListeResponse">
            <wsdlsoap:body
                encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                namespace="http://myriam:8080/axis/ServiceProvince.jws"
                space="http://myriam:8080/axis/ServiceProvince.jws"
                use="encoded" />
        </wsdl:output>
    </wsdl:operation>
    <wsdl:operation name="getChefLieu">
        <wsdlsoap:operation soapAction="" />
        <wsdl:input name="getChefLieuRequest">
            <wsdlsoap:body
                encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                namespace="http://DefaultNamespace" use="encoded" />
        </wsdl:input>
        <wsdl:output name="getChefLieuResponse">
            <wsdlsoap:body
                encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                namespace="http://myriam:8080/axis/ServiceProvince.jws"
                space="http://myriam:8080/axis/ServiceProvince.jws"
                use="encoded" />
        </wsdl:output>
    </wsdl:operation>
</wsdl:binding>
```

```

<wsdl:service name="ServiceProvinceService">
    <wsdl:port binding="impl:ServiceProvinceSoapBinding"
        name="ServiceProvince">
        <wsdlsoap:address
            location="http://myriam:8080/axis/ServiceProvince.jws" />
    </wsdl:port>
</wsdl:service>
</wsdl:definitions>

```

Il n'est pas très difficile de décoder ce document et d'en déduire la structure classique d'un document WSDL :

<wsdl:definitions>	tag racine obligatoire de tout document WSDL valide : il contient les noms des définitions du service (ici, ServiceProvince.jws – nous y reviendrons)
<wsdl:types> ... </wsdl:types>	définition, au moyen d'un élément schema, des types personnalisés utilisés par le service
<wsdl:message name="xxx"> <wsdl:part name="yyy" type="zzz" /> </wsdl:message>	pour définir le nom d'une méthode du service – les tags <part> permettent évidemment de définir la nature du(des) paramètre(s) de cette méthode
<wsdl:portType name="xxx"> <wsdl:operation name="yyy" ... <wsdl:input message="aaaRequest" name="xxxRequest" /> ... </wsdl:operation> ... </wsdl:portType>	permet de mettre en rapport une opération-méthode avec les messages associés (par exemple, la méthode getChefLieu est mise en rapport avec les messages getChefLieuRequest et getChefLieuResponse).
<wsdl:binding ...> <wsdlsoap:binding style="rpc" transport= "http://schemas.xmlsoap.org/soap/http" /> <wsdl:operation name="xxx"> <wsdl:input name="xxxRequest"> <wsdlsoap:body ...> </wsdl:input> ... </wsdl:operation> </wsdl:binding ...>	établit la relation entre chacune des opérations définies dans <portType> et le protocole physique : classiquement, il s'agit d'un appel de type RPC avec un transport par http

<pre><wsdl:service name="ServiceProvinceService"> <wsdl:port binding="xxx" name="yyy"> <wsdlsoap:address location="http://machine:port/zzz" /> </wsdl:port> </wsdl:service> </wsdl:definitions></pre>	établit la relation entre les éléments de liaisons et une adresse spécifique (ici, une URL de HTTP)
---	---

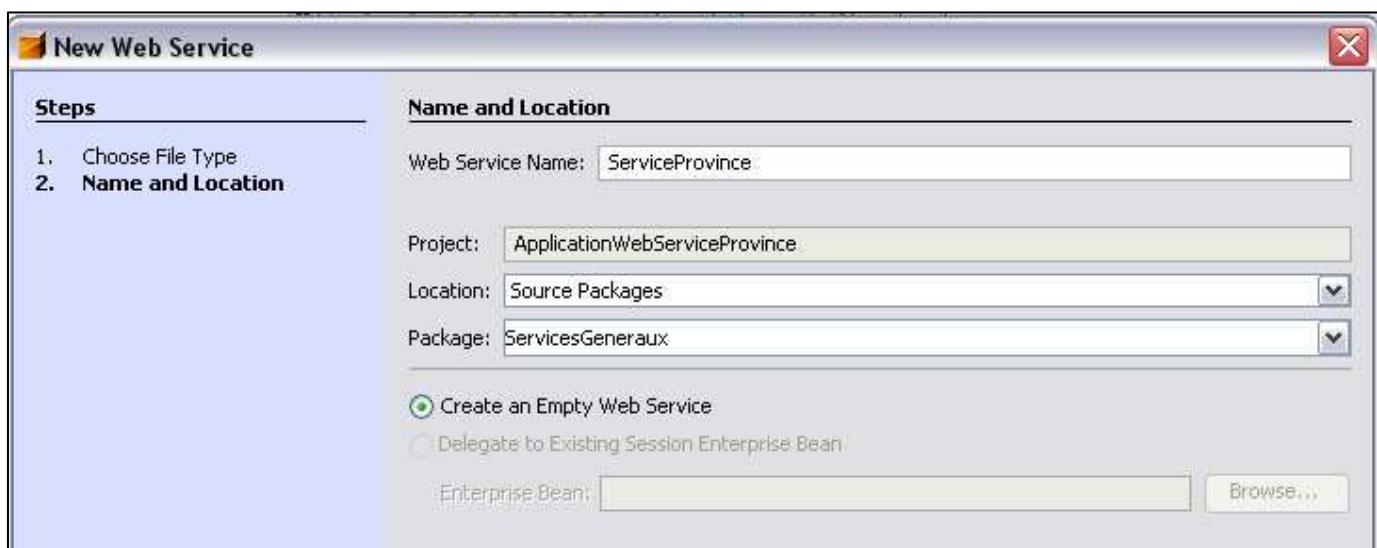
Bien sûr, la question dévore le lecteur : concrètement ?

5. Le déploiement d'un Web service sous NetBeans

Le déploiement d'un Web service sous Netbeans, ainsi que d'un client utilisant ce service, est remarquablement ais ,   condition de travailler en JDK 1.5. Que l'on en juge ☺ ...

5.1 La cr ation du Web Service

Nous cr ons une application Web des plus classiques (disons "ApplicationWebServiceProvince") et nous fixons l'utilisation du JDK 1.5 (dans les Properties du projet, non seulement avec l'onglet Librairies, clause "Java Platform", mais aussi avec l'onglet Sources, clause "Source Level"). Puis nous choisissons New → Web Service :



Nous obtenons un squelette de Web service (avec des annotations "@Web") :

ServiceProvince.java (code g n r )

```
/*
 * ServiceProvince.java
 *
 * Created on 26 ao t 2007, 12:15
 *
 * To change this template, choose Tools | Template Manager
 * and open the template in the editor.
 */

```

```
package ServicesGeneraux;
```

```
import javax.jws.WebMethod;  
import javax.jws.WebParam;  
import javax.jws.WebService;
```

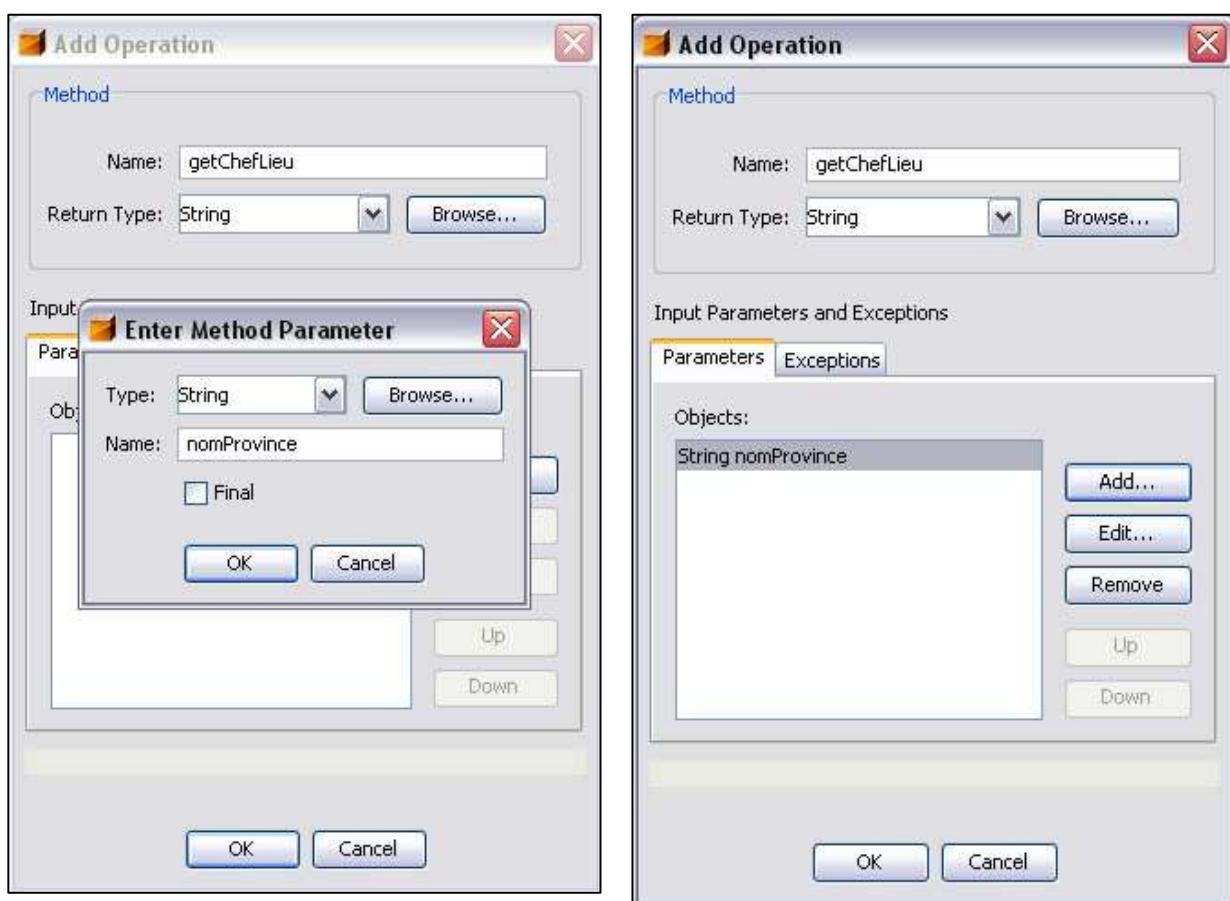
```
/**  
 *  
 * @author Vilvens  
 */
```

```
@WebService()  
public class ServiceProvince  
{  
}
```

Un clic droit sur le nœud "ServiceProvince" correspondant au Web service, également apparu dans sous le nœud "Web Services" :



fait apparaître un menu contextuel dans lequel nous choisissons "Add operation" – l'opération sera donc pour nous "getChefLieu", agrémentée d'un paramètre représentant le nom de la province :



Il nous reste à compléter selon ce qui a été dit ci-dessus pour obtenir finalement :

ServiceProvince.java (code modifié)

```
/*
 * ServiceProvince.java
 *
 * Created on 26 août 2007, 12:15
 */

package ServicesGeneraux;

import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebService;

import java.util.*;
import java.io.*;

/**
 * @author Vilvens
 */

@WebService()

```

```

public class ServiceProvince
{
    static private String FICHIER_PROVINCES_CHEFS_LIEUX =
        "c:\\java-netbeans-application\\ApplicationWebServiceProvince\\provinces-chefs-
        lieu.properties";
    Properties pCl;

    public ServiceProvince()
    {
        pCl = new Properties();
        try
        {
            pCl.load(new FileInputStream(FICHIER_PROVINCES_CHEFS_LIEUX));
        }
        catch (IOException e)
        {
            System.out.println("Erreur de lecture du fichier properties : " + e.getMessage());
        }
    }

    /**
     * Web service operation
     */
    @WebMethod
    public String getChefLieu(@WebParam(name = "nomProvince") String nomProvince)
    {
        String res = pCl.getProperty(nomProvince);
        if (pCl==null) return "Province inconnue";
        else return res;
    }
}

```

5.2 Le déploiement du Web Service

Après un "Deploy Project" sur le nœud de l'application, un nouveau clic droit sur le nœud "ServiceProvince" correspondant au Web service :



fait apparaître à nouveau le menu contextuel dans lequel nous choisissons cette fois "Test Web Service" : ceci revient à déployer notre Web Service, comme un page Web apparue dans le browser par défaut nous le signale (Tomcat attend sur son port habituel) :

Port Name	Status	Information
ServiceProvince	ACTIVE	Address: http://localhost:8084/ApplicationWebServiceProvince/ServiceProvince WSDL: http://localhost:8084/ApplicationWebServiceProvince/ServiceProvince?wsdl Port QName: (http://ServicesGeneraux/) ServiceProvincePort Implementation class: ServicesGeneraux.ServiceProvince

Un fichier WSDL a bien sûr été généré (dans le répertoire C:\java-netbeans-application\ApplicationWebServiceProvince\build\generated\wsgen\service) :

ServiceProvinceService.wsdl

```

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<definitions targetNamespace="http://ServicesGeneraux/" name="ServiceProvinceService"
  xmlns:tns="http://ServicesGeneraux/" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns="http://schemas.xmlsoap.org/wsdl/">

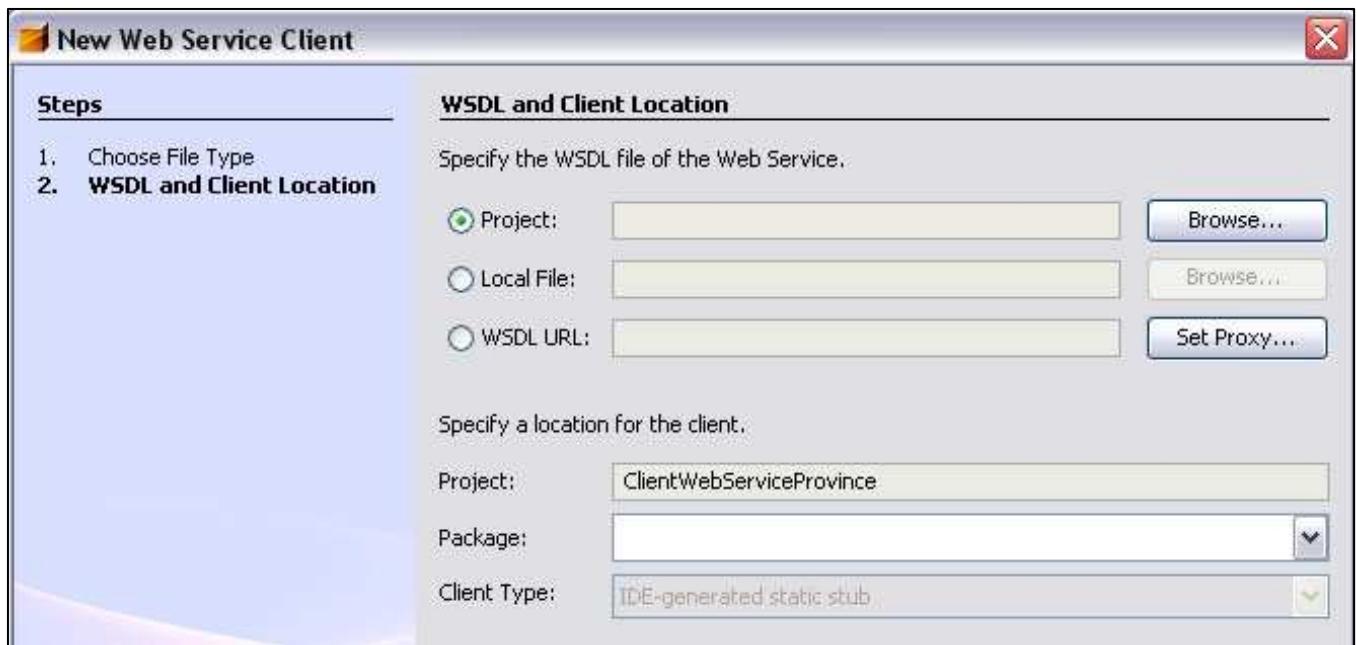
  <types>
    <xsd:schema>
      <xsd:import namespace="http://ServicesGeneraux/" schemaLocation="ServiceProvinceService_schema1.xsd"/>
    </xsd:schema>
  </types>
  <message name="getChefLieu">
    <part name="parameters" element="tns:getChefLieu"/>
  </message>
  <message name="getChefLieuResponse">
    <part name="parameters" element="tns:getChefLieuResponse"/>
  </message>
  <portType name="ServiceProvince">
    <operation name="getChefLieu">
      <input message="tns:getChefLieu"/>
      <output message="tns:getChefLieuResponse"/>
    </operation>
  </portType>
  <binding name="ServiceProvincePortBinding" type="tns:ServiceProvince">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
    <operation name="getChefLieu">
      <soap:operation soapAction="" />
      <input>
        <soap:body use="literal"/>
      </input>
    </operation>
  </binding>
</definitions>

```

```
</input>
<output>
  <soap:body use="literal"/>
</output>
</operation>
</binding>
<service name="ServiceProvinceService">
  <port name="ServiceProvincePort" binding="tns:ServiceProvincePortBinding">
    <soap:address location="REPLACE_WITH_ACTUAL_URL"/>
  </port>
</service>
</definitions>
```

5.3 Le client du Web service

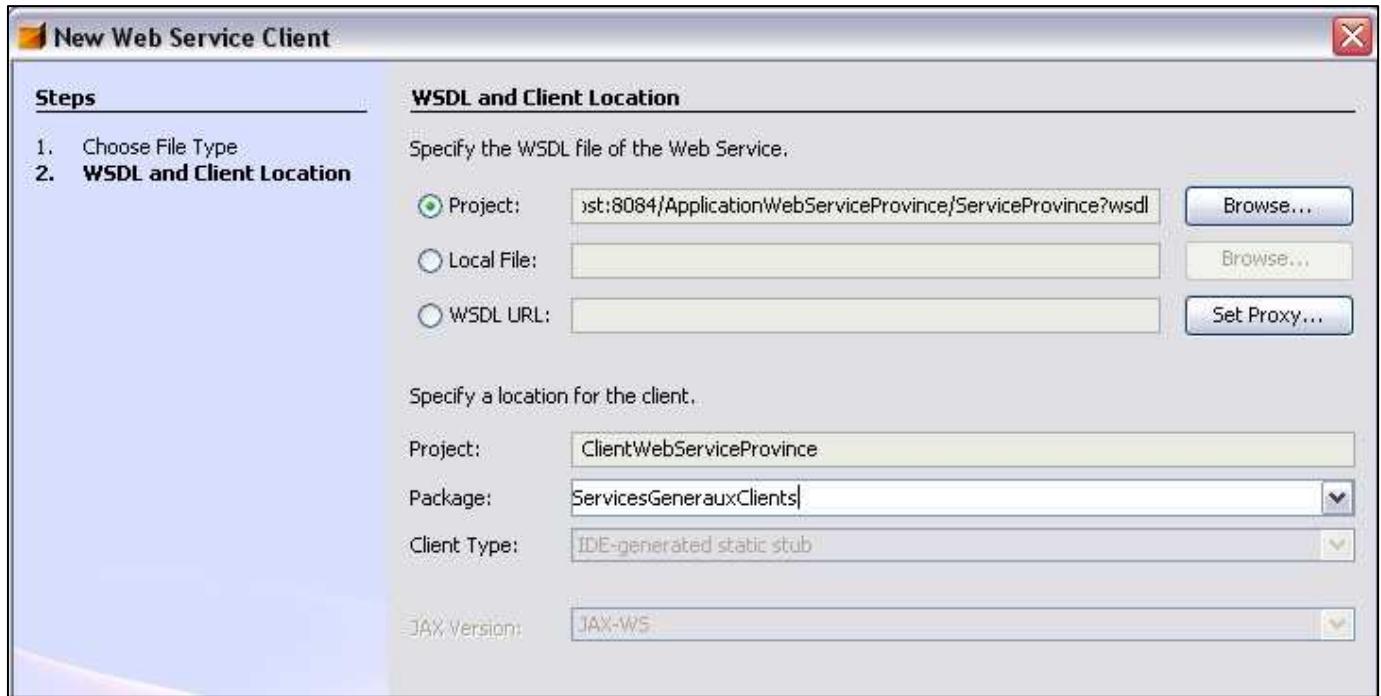
Maintenant que notre service Web est disponible, nous pouvons créer un client qui l'utilisera. Pour cela, nous créons une autre application web (disons "ClientWebServiceProvince" – toujours en JDK 1.5) et nous y créons un client par New → Web Service Client : nous parvenons ainsi à :



Il nous faut désigner le service Web visé, ce que nous faisons grâce à un appui sur le bouton "Browse" :

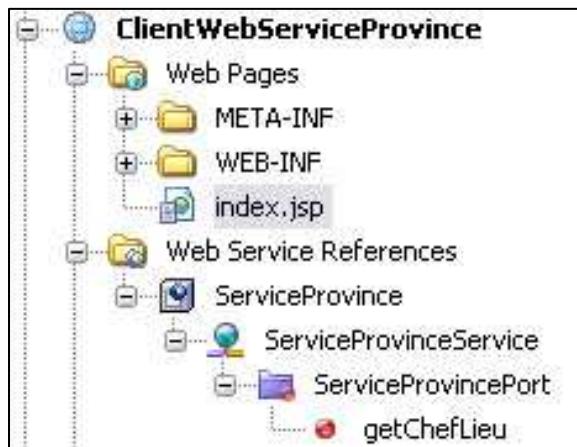


Donc, finalement :

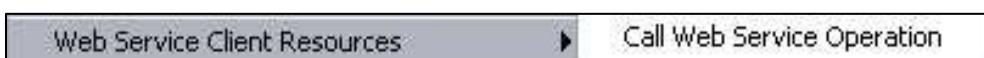


La référence au Web service est à présent enregistrée car le Web service a été retrouvé comme indiqué dans la fenêtre "retriever" ::

26-août-2007 14:45:36 : Retrieving Location:
http://localhost:8084/ApplicationWebServiceProvince/ServiceProvince?wsdl
Retrieved :



Notre client va exister, à priori, par l'intermédiaire d'un JSP nommé "index". Un clic droit dans son code permet de choisir dans un menu contextuel :



qui nous laisse choisir :



Le code ainsi généré correspond à une invocation type du Web service :

index.jsp (code généré)

```

<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>
<%-- The taglib directive below imports the JSTL library. If you uncomment it,
you must also add the JSTL library to the project. The Add Library... action
on Libraries node in Projects view can be used to add the JSTL 1.1 library.
--%>
<%-->
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
--%>

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

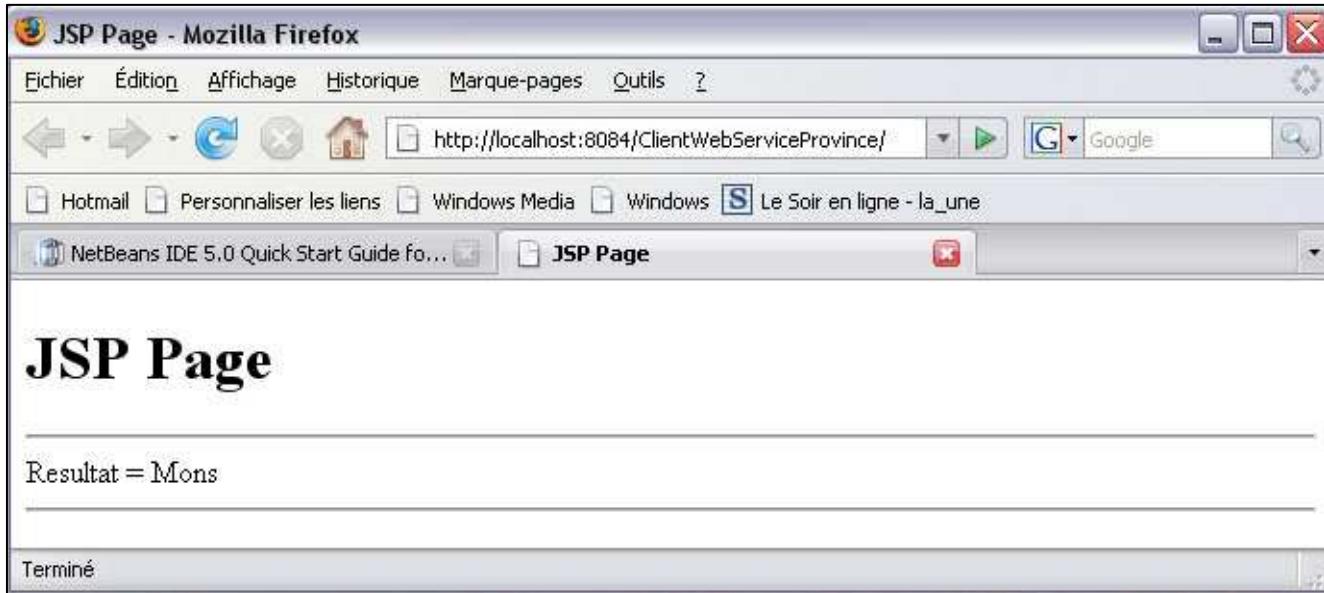
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>JSP Page</title>
  </head>
  <body>

    <h1>JSP Page</h1>

    <%-- start web service invocation --%><hr/>
    <%
    try {
      ServicesGenerauxClients.ServiceProvinceService service = new
        ServicesGenerauxClients.ServiceProvinceService();
      ServicesGenerauxClients.ServiceProvince port = service.getServiceProvincePort();
      // TODO initialize WS operation arguments here
      java.lang.String nomProvince = "Hainaut";
      // TODO process result here
      java.lang.String result = port.getChefLieu(nomProvince);
    }
  
```

```
    out.println("Resultat = "+result);
} catch (Exception ex) {
    // TODO handle custom exceptions here
}
%>
<%-- end web service invocation --%><hr/>
</body>
</html>
```

Sans finasser quoi que ce soit, l'exécution immédiate donnera :



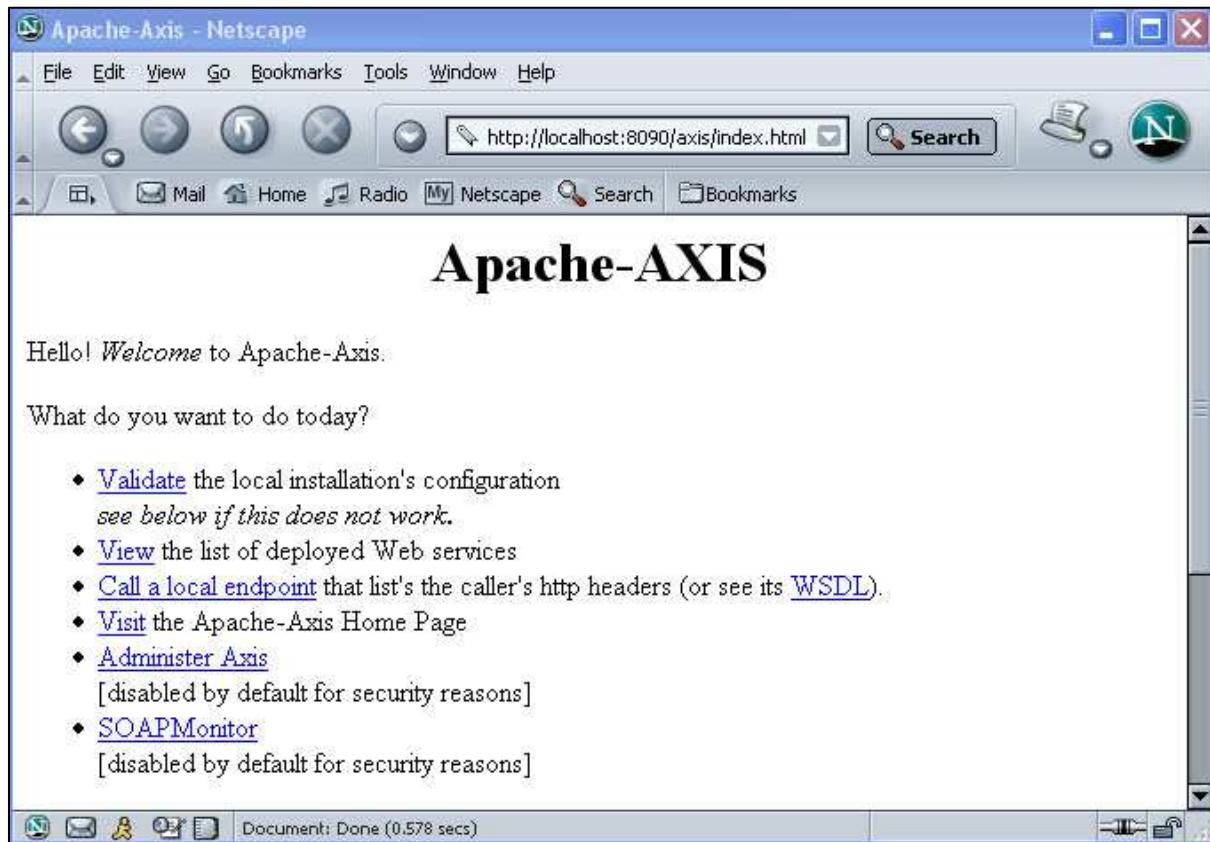
Il n'y a plus qu'à raffiner ... to do ;-)

6. Le déploiement d'un service selon Apache Axis

Afin d'illustrer une autre méthode de travail, nous allons utiliser ici comme plate-forme de déploiement (donc comme implémentation de SOAP) et de diffusion (donc comme serveur WSDL) des services Web l'environnement **Apache Axis** (Apache eXtensible Interaction System). C'est probablement l'un des plus simples à utiliser, même si il doit encore évoluer.

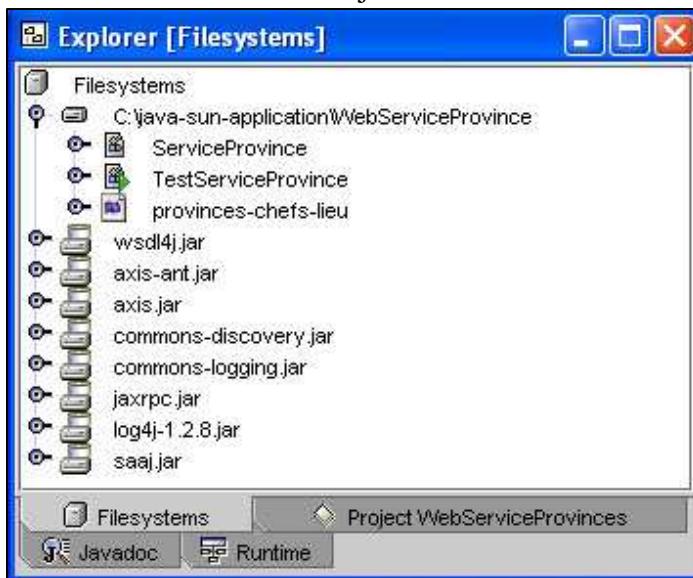
6.1 Le déploiement d'Apache Axis dans Tomcat

Le serveur Web est encore ici Tomcat. Y déployer Apache Axis est très simple : il suffit de recopier dans C:\Program Files\Apache Group\Tomcat 4.1\webapps le répertoire axis contenant les tous les éléments Axis nécessaires (les fichiers jar, le web.xml, etc). Mais alors ? Mais oui, c'est bien cela : **Apache Axis est une application Web** que l'on peut accéder avec un browser :



6.2 Le développement du service Web

Sous NetBeans, il suffit de créer dans un projet le fichier ServiceProvince.java évoqué en début de chapitre et de monter les fichiers jar d'Axis :



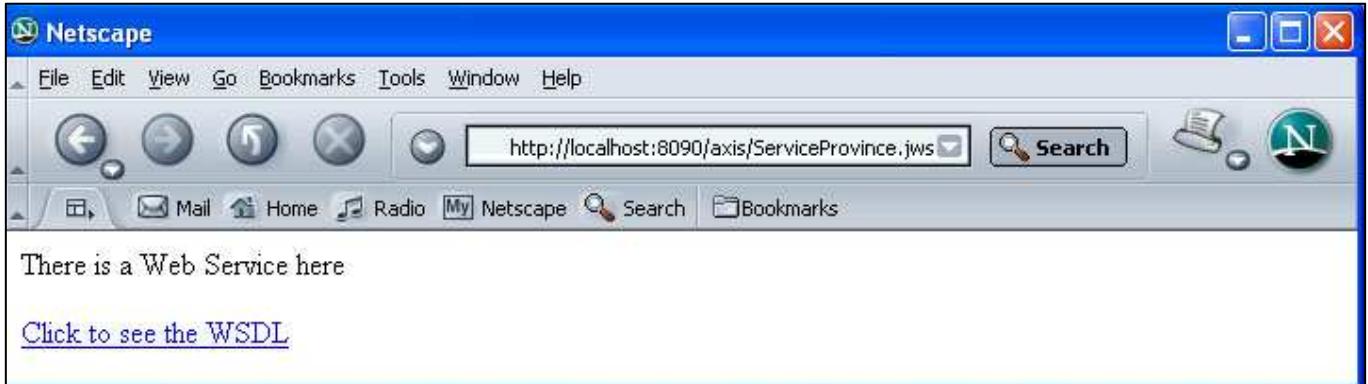
Après compilation histoire de vérifier que tout est en ordre, nous pouvons utiliser le "déploiement instantané" d'Axis qui consiste à :

- ◆ copier le fichier ServiceProvince.java dans C:\Program Files\Apache Group\Tomcat 4.1\webapps\axis;
- ◆ transformer l'extension ".java" de ce fichier en ".jws".

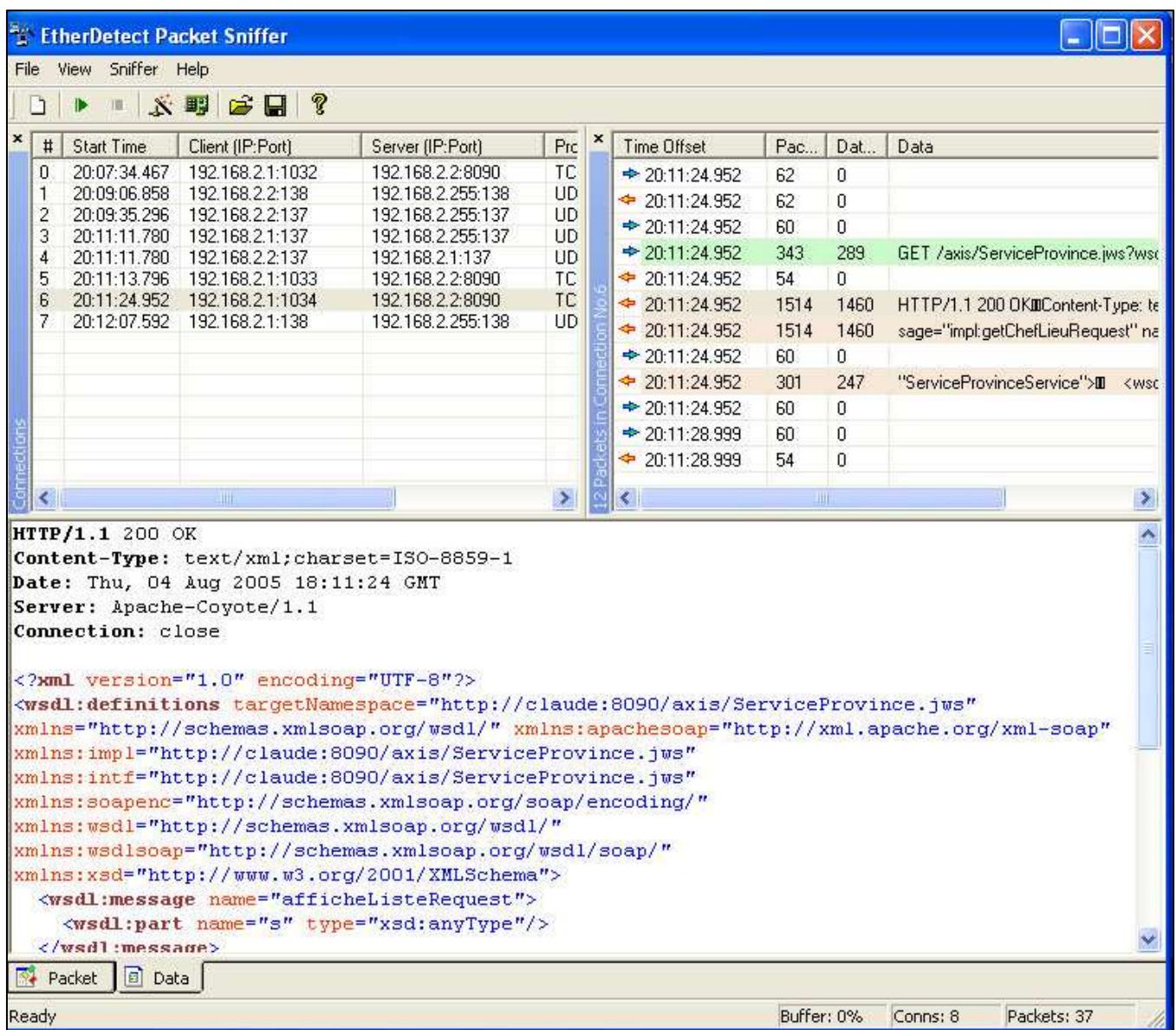
Miracle ! **Le service Web est prêt.** On peut le vérifier en utilisant l'URL :

http://localhost:8090/axis/ServiceProvince.jws
 ou
 http://192.168.2.2:8090/axis/ServiceProvince.jws

On obtient en effet :



et le transfert réseau peut être observé avec un sniffer :



L'aspect "**découverte du service**" est donc bien assuré avec le concours de http.

Mais que s'est-il réellement passé passé ? En fait, Axis a localisé le fichier jws demandé, l'a compilé et a placé le fichier ServiceProvince.class dans C:\Program Files\Apache Group\Tomcat 4.1\webapps\axis\WEB-INF\jwsClasses. Par quel sortilège ? Parce qu'une servlet d'Axis, nommée **AxisServlet**, est chargée, entre autres, de traiter les fichiers d'extension .jws. On peut le voir dans le web.xml de l'application Web qu'est Axis :

web.xml dans C:\Program Files\Apache Group\Tomcat 4.1\webapps\axis\WEB-INF

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app
PUBLIC "-//Sun Microsystems, Inc./DTD Web Application 2.2//EN"
"http://java.sun.com/j2ee/dtds/web-app_2.2.dtd">

<web-app>
    <display-name>Apache-Axis</display-name>
    <servlet>
        <servlet-name>AxisServlet</servlet-name>
        <display-name>Apache-Axis Servlet</display-name>
        <servlet-class>org.apache.axis.transport.http.AxisServlet
        </servlet-class>
    </servlet>
    ...
    <servlet-mapping>
        <servlet-name>AxisServlet</servlet-name>
        <url-pattern>/servlet/AxisServlet</url-pattern>
    </servlet-mapping>

    <servlet-mapping>
        <servlet-name>AxisServlet</servlet-name>
        <url-pattern>*.jws</url-pattern>
    </servlet-mapping>

    <servlet-mapping>
        <servlet-name>AxisServlet</servlet-name>
        <url-pattern>/services/*</url-pattern>
    </servlet-mapping>
    ....
</web-app>
```

Notre service est donc bien disponible. Reste à le solliciter ...

6.3 Le client du service Web

L'écriture d'un client Java voulant accéder à un service Web géré par Apache Axis est fort simple :

1) Il faut d'abord se fournir une instance de la classe **Call** (du package org.apache.axis.client). Le rôle de cette classe est de permettre les envois et réceptions de type RPC entre un client et un service Web à travers le moteur Axis. Le constructeur le plus usuel est :

```
public Call (String url) throws java.net.MalformedURLException
```

qui réclame évidemment l'URL du service demandé.

2) Si tout se passe bien, on peut alors appeler la méthode choisie (dans notre cas, getChefLieu()) au moyen de la méthode de Call :

```
public Object invoke(QName operationName, Object[] params)
                     throws java.rmi.RemoteException
```

le deuxième argument de cette méthode permettant évidemment de fournir le(s) paramètre(s) de la méthode du service que l'on veut invoquer. Dans notre cas, nous attendons comme réponse une chaîne de caractères si bien que le résultat sera casté en String.

C'est aussi simple que cela ☺ :

ClientProvince.java

```
/*
 * ClientProvince.java
 * Created on 26 avril 2004, 18:13
 */
/**
 * @author Vilvens
 */

import org.apache.axis.*;
import org.apache.axis.client.*;
import java.net.*; // pour MalformedURLException

public class ClientProvince
{
    public ClientProvince() { }

    public static void main(String[] args)
    {
        String urlService = "http://192.168.2.2:8090/axis/ServiceProvince.jws";
        Call client = null;
        try
        {
            client = new Call(urlService);
        }
        catch (MalformedURLException e)
        {
```

```
System.out.println("Erreur d'URL : " + e.getMessage());
System.exit(0);
}

Object[] requete = new Object[1];
requete[0] = new String("Hainaut");

String reponse = null;
try
{
    reponse = (String)client.invoke("getChefLieu", requete);
}
catch (AxisFault e)
{
    System.out.println("Erreur Axis : " + e.getMessage());
}

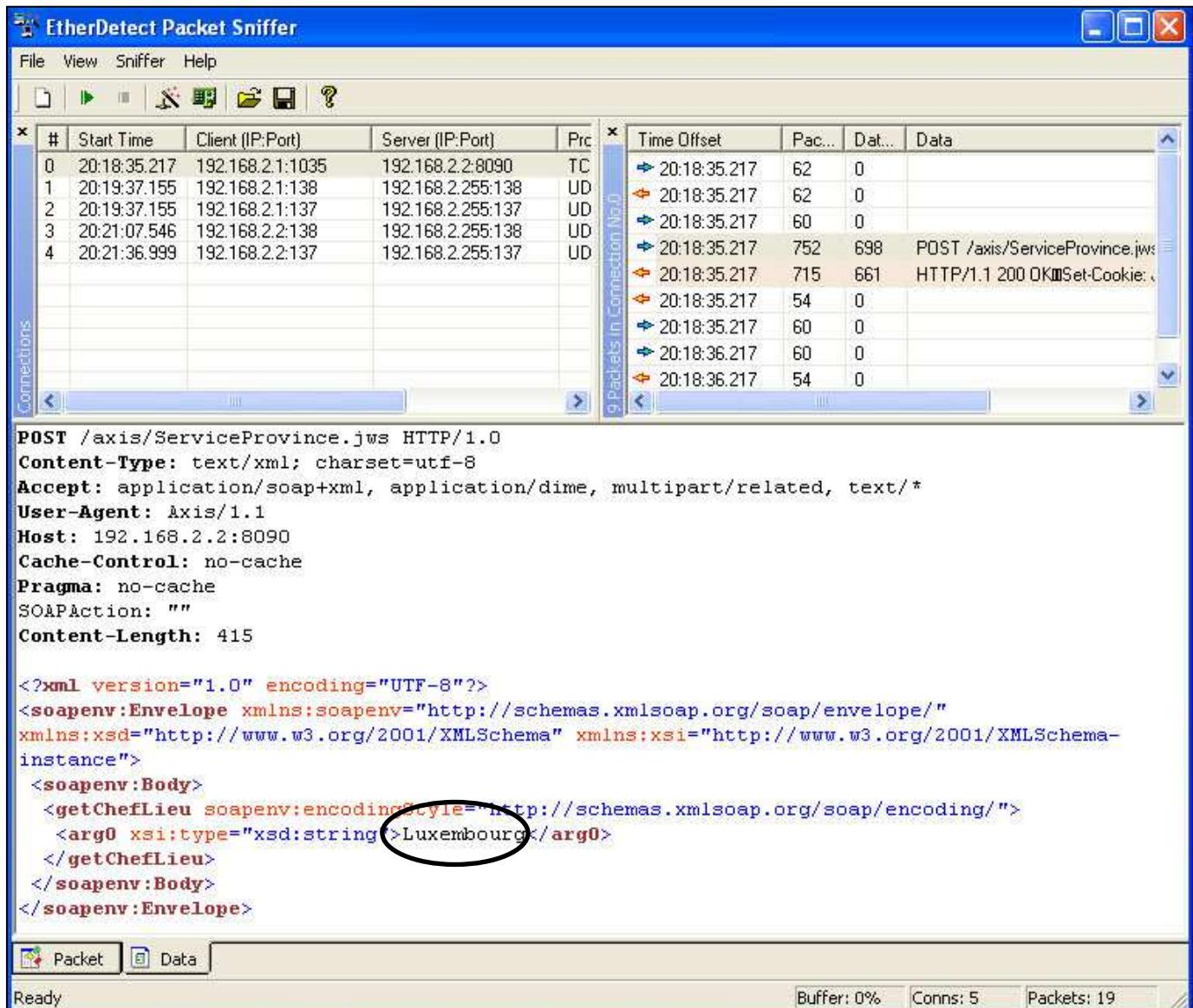
System.out.println("Résultat de la requête : " + reponse);
}
}
```

L'exécution de ce modeste client donnera bien dans la console :



Mais que s'est-il passé sur le réseau ? Bien évidemment, l'échange de messages SOAP supportés par http – le sniffer confirme :

a) la requête (ici, la question porte sur la province de Luxembourg) :



b) la réponse (woaw ! c'est juste ;-)) :

EtherDetect Packet Sniffer

File View Sniffer Help

Connections

#	Start Time	Client (IP:Port)	Server (IP:Port)	Pr...
0	20:18:35.217	192.168.2.1:1035	192.168.2.2:8090	TC
1	20:19:37.155	192.168.2.1:138	192.168.2.255:138	UD
2	20:19:37.155	192.168.2.1:137	192.168.2.255:137	UD
3	20:21:07.546	192.168.2.2:138	192.168.2.255:138	UD
4	20:21:36.999	192.168.2.2:137	192.168.2.255:137	UD

9 Packets in Connection No. 0

Time Offset	Pac...	Dat...	Data
20:18:35.217	62	0	
20:18:35.217	62	0	
20:18:35.217	60	0	
20:18:35.217	752	698	POST /axis/ServiceProvince.jws
20:18:35.217	715	661	HTTP/1.1 200 OK Set-Cookie: JSESSIONID=79288456199517131ABE39430FOFC2AB; Path=/axis
20:18:35.217	54	0	
20:18:35.217	60	0	
20:18:36.217	60	0	
20:18:36.217	54	0	

```

HTTP/1.1 200 OK
Set-Cookie: JSESSIONID=79288456199517131ABE39430FOFC2AB; Path=/axis
Content-Type: text/xml;charset=utf-8
Date: Thu, 04 Aug 2005 18:18:35 GMT
Server: Apache-Coyote/1.1
Connection: close

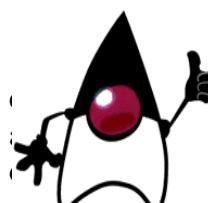
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
<ns1:instance>
<soapenv:Body>
<getChefLieuResponse soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<getChefLieuReturn xsi:type="xsd:string">Arlon</getChefLieuReturn>
</getChefLieuResponse>
</soapenv:Body>
</soapenv:Envelope>

```

Packet Data

Ready Buffer: 0% Conns: 5 Packets: 19

SOAP veillait bien sur nous ;-) !



Il y a encore à raconter concernant Java ! Outre les EJBs, tout juste faudrait s'intéresser aux classes de Java-mail, aux politiques de sécurité et trisées sans oublier SSL, aux Java-cards et aux mobiles, à d'autres protocoles SNMP, etc.

Peut-être dans un volume IV ? Sans doute : mais à chaque cours suffit sa peine - bye !

à suivre ...

Ouvrages consultés

Ouvrages imprimés

Akif, M, et al. Java XML – Programmer's Reference. Birmingham, United Kingdom. Wrox Press Ltd. 2001.

Armstrong, E. et al.. The Java Web Services Tutorial / The Java Series. Reading, Massachusetts, U.S.A. Addison-Wesley Publishing Company. 2002.

Arnold, K. & Gosling, J. The Java Programming Language - Second Edition / The Java Series. Reading, Massachusetts, U.S.A. Addison-Wesley Publishing Company. 1997.

Avedal, K, et al. Professional JSP. Birmingham, United Kingdom. Wrox Press Ltd. 2000.

Baguette, B. Java distributed systems : RMI & JINI. Seraing, Belgique. TFE In.Pr.E.S. 2002.

Bergsten, H. Java Server Pages. Sebastopol, California, U.S.A. O'Reilly and Associates, Inc. 2001.

Campione, M., Walrath, K., Huml, A & the tutorial team. The Java Tutorial Continued / The Java Series. Reading, Massachusetts, U.S.A. Addison-Wesley Publishing Company. 1998.

Degey, D. Etude de technologies Intranet/Internet dans un contexte pédagogique : Administration avec sécurisation de serveurs Web et communications entre applicatifs Java. Seraing, Belgique. TFE In.Pr.E.S. 2001.

Deneumostier, M. Etude des APIs d'extension Java : les Java Server Pages et le moteur Tomcat. TFE In.Pr.E.S. 2002.

Englander, R. Java and SOAP. Sebastopol, California, U.S.A. O'Reilly and Associates, Inc. 2002.

Flanagan, D. Java in a nutshell. Sebastopol, California, U.S.A. O'Reilly and Associates, Inc. 1999.

Flanagan, D., Farley, J., Crawford, W. & Magnusson, K. Java Enterprise in a nutshell. Sebastopol, California, U.S.A. O'Reilly and Associates, Inc. 1999.

Grégoire, A. Application des EJB pour du service Provisionning sur réseaux ATM. Seraing, Belgique. TFE In.Pr.E.S. 2001.

Harold, E.R. & Means, W.S. XML in a nutshell. Sebastopol, California, U.S.A. O'Reilly and Associates, Inc. 2001.

Hunter, J. & Crawford, W. Java servlets. Sebastopol, California, U.S.A. O'Reilly and Associates, Inc. 1998.

Klein, D. Etude des Enterprise Java Beans. Seraing, Belgique. TFE In.Pr.E.S. 2002.

Langlet, E. Apache Tomcat 5 / Serveur d'applications Java : Administration sous Windows et Linux. St Herblain, France. Ed. ENI, collection Ressources informatiques, 2006.

Leburton, C. Utilisation des technologies Java et XML dans le cadre d'un projet européen : Onto-Logging. TFE In.Pr.E.S. 2002.

Monson-Haefel R. Enterprise JavaBeans. Sebastopol, California, U.S.A. O'Reilly and Associates, Inc. 1999.

Reip, S. Etude de diverses implémentations des services Web sous Java. Seraing, Belgique. TFE In.Pr.E.S. 2004.

Robaux, Y. Etude du middleware Jonas dans le cadre de la faisabilité d'évolution de l'application SoftFM vers une architecture distribuée J2EE. TFE ISIL. 2004.

Vilvens, C. Langage Java (I) : Programmation de base. Seraing, Belgique. A.S.B.L. DEFI. 2010.

Vilvens, C. Langage Java (II) : Programmation avancée des applications classiques et cryptographie. Seraing, Belgique. A.S.B.L. DEFI. 2011.

Vilvens, C. Technologies Web élémentaires : HTTP, HTML, CGI, Javascript et Ajax. Seraing, Belgique. A.S.B.L. DEFI. 2011.

Sites Internet

<http://java.sun.com/>

avec en particulier :

<http://developer.java.sun.com/>

et

<http://developer.java.sun.com/developer/onlineTraining/>

<http://www.bejug.org/>

Site du Belgian Java User Group

<http://sunsite.cnlab-switch.ch/>

Site de Sun en Suisse

http://java.sun.com/developer/technicalArticles/J2EE/intro_ee5/

Update: An Introduction to the Java EE 5 Platform

<http://java.sun.com/developer/onlineTraining/rmi/RMI.html>

A RMI short course (jGuru): Remote Method Invocation (RMI)

<http://psecheresse.developpez.com/tutoriel/netbeans/java-ee/secured/>

Introduction au développement des EJB 3.0 sous Netbeans 6

<http://www.netbeans.org/kb/60/websvc/jax-ws.html>

Getting Started with JAX-WS Web Services



Annexe : Les bases de XML



*Mais les rêves, tous ces rêves
Que l'on ne faisait plus
Mais les rêves, tous ces rêves
Que l'on croyait perdus
Il suffit d'une étincelle
Pour que tout à coup
Ils reviennent de plus belle
Les rêves sont en nous*

(Pierre Rapsat, Les rêves sont en nous)

On a pu constater dans les différents chapitres du présent ouvrage que XML est omniprésent, que ce soit, évidemment, pour Java XML ou pour l'écriture des fichiers de configuration des Java Server Pages ou autres. Il est donc nécessaire de posséder un minimum de connaissances sur ce célèbre méta-langage – c'est le but de cette annexe. Bien sûr, un lecteur intéressé pourra se tourner vers les ouvrages spécialisés dévolus plus profondément au sujet (voir les ouvrages consultés).

1. SGML et XML

Le célèbre langage HTML est un langage spécifique écrit en respectant les spécifications de **SGML** (Standard Generalized Markup Language). En fait, SGML est un **méta-langage**, en ce sens qu'il permet de décrire des langages à *balises* ou *marqueurs*, ces dernières permettant de spécifier comment le texte doit être manipulé. Un tel langage à balises n'est pas un concept neuf : les typographes en connaissent depuis longtemps.

SGML est donc un standard international permettant de définir la représentation des textes électroniques indépendamment des machines et des systèmes d'exploitation sous-jacents. Il permet donc de définir les balises d'autres langages, leurs enchaînements de balises obligatoires, la manière de reconnaître ces balises. Plus précisément, un document suivant SGML comporte notamment une partie déclaration (elle spécifie, outre le jeu de caractères utilisés et les caractères qui ont une signification particulière, la structure de balises utilisée) et, bien sûr, le document proprement dit, avec ses balises.

HTML est en fait une application de SGML, qui possède sa description propre¹⁸ écrite en SGML.

SGML est cependant assez compliqué : ses spécifications s'étirent sur plus 150 pages. Pour cette raison, on a créé un méta-langage sous-ensemble de SGML spécialement dirigé vers les applications WEB : il se nomme **XML** (eXtensible Markup Language). Ses spécifications ne couvrent qu'environ 35 pages¹⁹.

A chaque fois que l'on crée un document XML, en définissant ses propres balises, *on crée en fait un nouveau langage selon des règles qu'il conviendra de connaître pour utiliser le document*. Fondamentalement, l'objectif est de structurer l'information selon des conventions que tous les utilisateurs intéressés connaîtront et appliqueront. L'information peut être de nature très diverse : des feuilles de calcul, des carnets d'adresses, des paramètres de configuration, des transactions financières, des dessins techniques, etc. Bien qu'initialement

¹⁸ nous pourrions dire sa DTD, en anticipant sur les explications qui vont suivre.

¹⁹ <http://www.w3.org/TR/REC-xml>

prévu seulement pour les applications Internet, XML a rapidement gagné d'autres médias d'informations. Le **W3C**, qui a approuvé les spécifications XML, le définit d'ailleurs comme :

une syntaxe commune permettant d'exprimer des structurations des données.

W3C gère donc les documents de référence de XML comme les projets futurs. Une autre organisation, **OASIS** (Organization for the Advancement of Structured Information Standards), gère quant à elle le "XML industry portal"²⁰ et la mailing list xml-dev-links.

Signalons l'existence de XHTML : il s'agit d'une reformulation de HTML mais avec une description de structure²¹ en XML. L'intérêt est de pouvoir utiliser pour HTML les outils plus légers de XML au lieu de ceux de SGML.

Comme en HTML, une balise [tag] se reconnaît au fait qu'elle débute par "<" et se termine par ">". Mais encore ?

2. Les balises XML

La forme d'un document XML rappelle fortement celle d'un document HTML rédigé "proprement". Ainsi, les constituants de base sont les "éléments", c'est-à-dire une suite formée d'une balise de départ [**start tag**] et d'une balise de fin [**end tag**], celle dernière étant caractérisée par la présence d'un "/" avant ">"; éventuellement, un texte, qui est en fait l'information en elle-même, peut se trouver entre ces deux balises – on l'appelle le **CDATA** (*Character DATA*). Par exemple :

<**invite**>Bonjour chers amis !!</**invite**>

La grosse différence par rapport à HTML, c'est que c'est l'utilisateur qui a défini le tag "invite" au lieu d'utiliser un tag prédefini ! Pour le reste, XML connaît

- ◆ les tags qui n'encadrent pas un texte (on parle encore d'"*empty element*") : ils n'ont pas besoin d'un tag de fin, comme en HTML, mais XML réclame d'eux qu'ils se terminent par "/>" au lieu du simple ">" de HTML – par exemple :

<**newpage**> </**newpage**>

peut être abrégé par :

<**newpage**/>

- ◆ la notion d'**attribut** pour un tag, c'est-à-dire une information additionnelle décrivant l'information; cette information est fournie sous la forme d'une paire nom=valeur; la valeur doit cependant être toujours entourée de guillemets; un attribut est évidemment extrêmement utile pour les éléments vides, comme par exemple :

<**logo** hauteur="12" épaisseur="3" écart="5" />

²⁰ http://www.xml.org/xml/news_market.shtml

²¹ encore une fois, en anticipant, sa DTD

- ◆ l'utilisation des **entités** [*entities*] qui rappellent les entités de caractères d'HTML : il s'agit donc d'un nom donné à un caractère spécial. Par exemple, un guillemet est connu par son nom, "quot", placé entre un caractère particulier ('&') qui en marque le début et un autre (';') qui en marque la fin (donc, ici, par "). Cinq entités sont ainsi prédéfinies :

symbole	entité
"	"
'	'
<	<
>	>
&	&

- ◆ les **commentaires**, encadrés par "<!--" et "-->" ;
- ◆ la distinction entre les minuscules et les majuscules [case sensitive]; la balise de prologue <?xml ...> est donc bien différente de <?XML ...>.
- ◆ les **instructions exécutables** [*processing instructions = PI*] qui sont de la forme :

<? cible instructions ?>

Dans le cas du tag <?xml ...?>, les instructions sont destinées au programme de traitement xml (ce que l'on appelle le parser). Dans les autres cas, les instructions sont transmises telles quelles à l'application. De telles PIs peuvent se trouver en un endroit quelconque du document.

Remarque

Les entités permettent donc d'ignorer la signification habituelle des symboles de XML comme "<" ou ">". Lorsque ceci concerne une zone de texte assez important, il est plus pratique de définir une section CDATA, délimitée par :

```
<![CDATA[ ....
....]
]]>
```

Tout ce qui se trouve dans une telle section sera traité comme du texte brut, sans tentative d'interprétation.

3. La syntaxe de base d'un document XML

Un document XML comporte un **prologue** [*prolog*] contenant au minimum la déclaration XML <?xml ...?> et un **corps** constituant le document proprement dit. Cependant, celui-ci doit toujours comporter un **élément racine** [*root element*], c'est-à-dire un élément donc les tags de début et de fin entourent l'ensemble du document. Donc, schématiquement, si le tag racine s'appelle par exemple "sommaire" :

```
<?xml ...?>
```

```
<sommaire>
```

```
...
```

```
</sommaire>
```

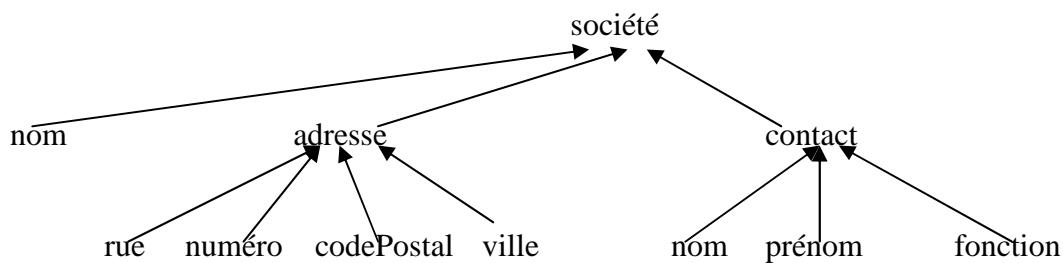
Les attributs du tag de prologue peuvent être :

- ◆ la version (obligatoire) : `version="1.0"`
- ◆ le fait pour le document de définir lui-même tous ses tags ou pas (certains sont alors définis dans un document externe, notamment une DTD) : `standalone="yes"`
- ◆ le schéma d'encodage des caractères : `encoding="UTF-8"`.

Donc, par exemple :

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
```

Les différents éléments du document sont impérativement imbriqués les uns dans les autres, sur base des tags de début et de fin (comme dans les langages de programmation). Ceci reflète aussi **l'arborescence sous-jacente de l'information traitée** [tree]. Ainsi, dans un contexte de communication d'une société de services (par exemple) :



on élaborera un document xml du type suivant :

société.xml

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<listeSociétés>
    <société>
        <nom>Genius S.A.</nom>
        <adresse>
            <rue>rue de la Réussite</rue>
            <numéro>113</numéro>
            <codePostal>4500</codePostal>
            <ville>Visé</ville>
        </adresse>
        <contact>
            <nom>Bidendum</nom>
            <prénom>Albert</prénom>
            <fonction>General Manager</fonction>
        </contact>
    </société>
</listeSociétés>
```

A ce stade, ce document est syntaxiquement correct. Cependant, rien ne permet de savoir s'il respecte des règles connues et donc s'il peut être utilisé par un utilisateur autre que son auteur. Nous allons y revenir.

4. Les parsers et leurs modèles

L'opération appelée "parsing" consiste

- ◆ à convertir une séquence XML en composants structurés et
- ◆ à entamer l'une ou l'autre action en fonction des composants détectés.

Une action logiquement attendue est simplement de fournir les informations reconnues à une application.

L'application capable de réaliser un tel traitement sur des fichiers XML est bien normalement appelée un "**parser**" ("parseur" en français ?). Bien sûr, un parser doit, normalement, détecter toute malformation du document XML "*parcé*" et signaler dans ce cas une erreur à l'application consommatrice. A priori, le parser interrompt alors son traitement; on peut cependant parfois paramétrer le parser pour qu'il poursuive malgré tout son travail. Si, de plus, la validation par rapport à une DTD (voir ci-dessous) est prise en charge, on parle encore parfois de **parser-validator** (parseur-validateur).

Deux modèles de base pour les parsers existent :

- ◆ **SAX (Simple Api for Xml)** est un ensemble d'APIs construit sur la notion d'événement. En fait, le document XML est lu séquentiellement et génère au fur et à mesure de la progression de cette lecture une suite d'événements du type : "début d'élément", "attribut d'un élément", "fin d'un élément", etc. Le document traité ne doit donc pas se trouver intégralement en mémoire, ce qui est intéressant pour les gros fichiers XML. SAX existe en version 2.0.
- ◆ **DOM (Document Object Model)** est un ensemble d'APIs construit sur la notion d'arbre. Le document à analyser est placé intégralement en mémoire et est mis sous la forme d'un arbre. Ce n'est qu'après cette analyse que les renseignements attendus sont fournis. L'inconvénient est évidemment que l'entièreté du document doit se trouver en mémoire. En sa version 2.0, DOM permet de ne placer en mémoire que la partie de l'arbre que l'on désire analyser.

Un parser très répandu, qui est aussi un validateur, est **Xerces**, fourni par Apache²² en open source. Il implémente les APIs SAX et DOM et existe pour Java, C++ et Perl.

5. La validité d'un document XML

5.1 Le contrôle d'un document XML : la DTD

Comment un browser peut-il utiliser un document XML ? Bien sûr, tel quel, il en est incapable puisque chacun a défini son propre langage ! Il aura donc besoin d'un parser qui

- vérifie que le document présente une structure XML correcte;
- extrait les données effectives de leur représentation textuelle en respectant leur structure.

Sur quelle base le parser peut-il décider qu'un document XML est valide ou pas ? En se basant sur les métainformations fournies dans le prologue par une **DTD (Document Type Definition/Declaration)** : il s'agit de la liste des tags valides, ainsi que de leurs inter-relations, liste exprimée selon une **syntaxe particulière non-xml**. La DTD peut se trouver dans le fichier XML lui-même ou dans un fichier externe (nous en reparlerons au paragraphe 9).

²² <http://xml.apache.org/>

5.2 La structure d'une DTD

Dans sa forme la plus simple, un DTD est intégré au fichier XML et se trouve dans une balise

```
<!DOCTYPE nom_structure_informations [  
...  
]>
```

exactement comme HTML procède pour signifier son niveau de conformité avec telle ou telle version du langage, comme par exemple

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 //EN">.
```

La DTD se résume ensuite, dans sa version la plus simple du moins, à une liste d'**ETD** (*Element Type Declaration*), c'est-à-dire des balises débutant par **<!ELEMENT**, citant le nom d'un élément et définissant les caractéristiques de cet élément. Celles-ci peuvent être :

- ◆ **#PCDATA** (entre parenthèses) : pour signifier que l'élément ne contient que du texte (*Parsed Character DATA*) après le tag (donc, pas d'autres éléments, comme ce serait le cas avec CDATA); on peut donc écrire, par exemple :

```
<!ELEMENT nom (#PCDATA)>
```

Ce texte est destiné à être parse et les entités doivent donc être utilisées, si nécessaire.

- ◆ **EMPTY** : pour signifier que le tag ne contient pas de texte (le fait d'être vide ne l'empêche pas d'avoir des attributs) - il s'utilisera donc sous la forme `<tag .../>`; par exemple :

```
<!ELEMENT importance EMPTY>
```

- ◆ un **ECM** (*Element Content Model*), c'est-à-dire une liste d'éléments enfants entre parenthèses; la virgule comme séparateur (en fait, le seul possible contrairement à SGML) implique que l'ordre d'énumération devra être respecté (on parle alors d'une "séquence"). Par exemple :

```
<!ELEMENT société (nom, adresse, contact)>
```

De plus, le nom de chaque élément enfant peut être suffixé d'un symbole indiquant le nombre de fois qu'il peut être répété :

symbole suffixe	signification
?	0 ou 1 fois
*	0 ou plusieurs fois
+	1 ou plusieurs fois
<rien>	1 et 1 seule fois

On peut donc écrire, par exemple :

```
<!ELEMENT société (nom, adresse+, contact*)>
```

- ◆ **ANY** : pour spécifier qu'un élément peut contenir une combinaison quelconque des éléments enfants (au sens de la hiérarchie implicite); ceci a surtout un sens pour le tag racine, car dans les autres cas, ceci conduit à une DTD assez vague; on peut donc imaginer par exemple :

```
<!ELEMENT dossier ANY>
```

- ◆ un choix entre différents éléments : le symbole "|" sépare les listes possibles de valeurs, comme par exemple dans :

```
<!ELEMENT note ( (expéditeur, contenu) | (expéditeur, destinataire+, contenu) )
```

A remarquer que si l'on veut exprimer que trois éléments A, B et C peuvent se présenter dans un ordre quelconque, on doit écrire :

```
A,B,C|A,C,B|B,A,C|B,C,A|C,A,B|C,B,A
```

si bien que l'on préfère le plus souvent un ordre arbitraire.

Notre DTD exemple peut donc être, dans sa forme la plus simple :

DTD listeSociétés

```
<!DOCTYPE listeSociétés [  
    <!ELEMENT listeSociétés ANY>  
    <!ELEMENT société (nomSociété, adresse, contact)>  
    <!ELEMENT nomSociété (#PCDATA)>  
    <!ELEMENT adresse (rue, numéro, codePostal, ville)>  
    <!ELEMENT rue (#PCDATA)>  
    <!ELEMENT numéro (#PCDATA)>  
    <!ELEMENT codePostal (#PCDATA)>  
    <!ELEMENT ville (#PCDATA)>  
    <!ELEMENT contact (nom, prénom, fonction)>  
    <!ELEMENT nom (#PCDATA)>  
    <!ELEMENT prénom (#PCDATA)>  
    <!ELEMENT fonction (#PCDATA)>  
>]
```

Il faut cependant remarquer que :

- ◆ une DTD n'est pas écrite en XML et ne peut donc être parsée par des APIs DOM ou SAX;
- ◆ les DTDs ne supportent pas les espaces de noms²³, si bien qu'il n'est pas possible en pratique d'importer des DTDs externes;
- ◆ le typage des données au sein d'une DTD est extrêmement limité.

Les schémas XML permettront de surmonter ces difficultés.

²³ les espaces de noms permettent de définir des tags XML sans crainte de collisions avec d'autres tags extérieurs – ils seront évoqués plus en détail au paragraphe 9.

6. Un document XML valide pour une DTD

Notre document XML, avec une DTD intégrée, peut donc à présent s'écrire :

listeSociétés.xml (3)

```
<?xml version="1.0" encoding="UTF-8"?>

<!—Created by Vilvens on 19 avril 2002, 17:00 -->
<!DOCTYPE listeSociétés [
    <!ELEMENT listeSociétés ANY>
    <!ELEMENT société (nomSociété, adresse, contact)>
    <!ELEMENT nomSociété (#PCDATA)>
    <!ELEMENT adresse (rue, numéro, codePostal, ville)>
    <!ELEMENT rue (#PCDATA)>
    <!ELEMENT numéro (#PCDATA)>
    <!ELEMENT codePostal (#PCDATA)>
    <!ELEMENT ville (#PCDATA)>
    <!ELEMENT contact (nom, prénom, fonction)>
    <!ELEMENT nom (#PCDATA)>
    <!ELEMENT prénom (#PCDATA)>
    <!ELEMENT fonction (#PCDATA)>
]>
<listeSociétés>
    <société>
        <nomSociété>Genius S.A.</nomSociété>
        <adresse>
            <rue>rue de la Réussite</rue>
            <numéro>113</numéro>
            <codePostal>4500</codePostal>
            <ville>Visé</ville>
        </adresse>
        <contact>
            <nom>Bidendum</nom>
            <prénom>Albert</prénom>
            <fonction>General Manager</fonction>
        </contact>
    </société>
</listeSociétés>
```

Mais nous pouvons apporter des améliorations. Supposons ainsi que nous ajoutions dans l'élément contact l'adresse e-mail :

```
...
    <!ELEMENT contact (nom, prénom, fonction, email)>
    <!ELEMENT nom (#PCDATA)>
    <!ELEMENT prénom (#PCDATA)>
    <!ELEMENT fonction (#PCDATA)>
    <!ELEMENT email (#PCDATA)>
...
...
```

si bien que le document XML peut ressembler à ceci :

```
...
<contact>
    <nom>Bidendum</nom>
    <prenom>Albert</prenom>
    <fonction>General Manager</fonction>
    <email>abidend@groumf.com</email>
</contact>
...
```

Mais une personne donnée peut très bien ne pas avoir d'e-mail ou, au contraire, en avoir plusieurs ! Nous en tenons compte en utilisant le symbole * :

```
...
<!ELEMENT contact (nom, prénom, fonction, email*)>
<!ELEMENT nom (#PCDATA)>
<!ELEMENT prénom (#PCDATA)>
<!ELEMENT fonction (#PCDATA)>
<!ELEMENT email (#PCDATA)>
...
...
```

si bien que le document XML peut alors aussi ressembler à ceci :

```
...
<contact>
    <nom>Bidendum</nom>
    <prenom>Albert</prenom>
    <fonction>General Manager</fonction>
    <email>abidend@groumf.com</email>
    <email>albert@sexy.net</email>
</contact>
...
```

Mais si le contact n'a pas d'e-mail, il a peut-être un numéro de téléphone :

```
...
<!ELEMENT contact (nom, prénom, fonction, (telephone | email*))>
<!ELEMENT nom (#PCDATA)>
<!ELEMENT prénom (#PCDATA)>
<!ELEMENT fonction (#PCDATA)>
<!ELEMENT telephone (#PCDATA)>
<!ELEMENT email (#PCDATA)>
...
...
```

Le nouveau document XML peut à présent avoir l'aspect suivant :

```

... <contact>
    <nom>Bidendum</nom>
    <prenom>Albert</prenom>
    <fonction>General Manager</fonction>
    <telephone>32(0)4 3307599</telephone >
</contact> ...

```

On constate donc que la DTD permet de définir (et de vérifier) un langage particulier à l'application envisagée : c'est en ce sens qu'il faut comprendre que XML est un méta-langage : **XML permet de définir d'autres langages.**

7. Les attributs dans les DTD

Les tags XML, comme les tags HTML, peuvent être complétés d'attributs. Une DTD doit donc également donner des informations sur ceux-ci afin de permettre l'évaluation de la validité du document qui prétend l'utiliser. C'est le rôle des balises débutant par **<!ATTLIST**, citant le nom d'un élément et définissant les caractéristiques des attributs possibles de cet élément. Pour chaque attribut, on trouve :

- ◆ son **nom**;
- ◆ son **type**; le plus courant est **CDATA**, déjà rencontré, signifiant qu'il peut s'agir d'une chaîne de caractères quelconque; un autre type fort utile est **ID**, signifiant que cet attribut doit être unique dans le document; on peut aussi citer toutes les valeurs possibles d'un attribut dans une énumération dont les éléments sont séparés par "|".

Donc, par exemple :

```

<!ELEMENT importance EMPTY>
<!ATTLIST importance notification CDATA>

<!ELEMENT numeroEnregistrement (#PCDATA)>
<!ATTLIST numeroEnregistrement ONSS ID>

<!ELEMENT commande (typeaction)>
<!ATTLIST commande type (add | remove | list) >

```

Quelques types possibles sont brièvement repris ci-dessous :

mot réservé	valeur
CDATA	texte quelconque mais valide pour un attribut XML
ID	nom XML (donc pas un nombre) unique dans le document
IDREF	valeur faisant référence à un ID défini auparavant – par exemple : <!ELEMENT personne (#PCDATA)> <!ATTLIST personne numero_ONSS ID #REQUIRED> <!ELEMENT membre_service (#PCDATA)> <!ATTLIST membre_service personne IDREF #REQUIRED implique que le dernier attribut ne peut prendre comme valeur qu'une valeur définie pour l'attribut numero_ONSS.
IDREFS	liste d'IDREF séparés par des blancs
NMTOKEN	unité lexicale nominale, c'est-à-dire une chaîne formée de caractères admis pour un nom XML (ce qui exclut notamment les blancs)
NMTOKENS	liste d'unités lexicales séparées par des blancs
<énumération>	valeurs séparées par " "

- ♦ son caractère **obligatoire ou optionnel**, choisi parmi :

mot réservé	signification
#REQUIRED	obligatoire
#IMPLIED	optionnel
#FIXED	l'attribut possède une valeur par défaut (spécifiée dans la DTD après le mot) et c'est toujours cette valeur qui doit être utilisée pour que le document soit valide.

Cela peut donner, par exemple :

```
<!ELEMENT commande (typeaction)>
<!ATTLIST commande type (add | remove | list) #REQUIRED>
```

Donc, l'élément <commande> doit

- ♦ comporter un seul élément typeaction (qui devra bien sûr être défini lui-même, par exemple #PCDATA);
- ♦ être nécessairement d'un type précisé parmi les trois valeurs possibles add, remove ou list.

Un élément valide selon la portion de DTD qui le concerne sera alors :

```
<commande type="add">
    <typeaction> immediat </typeaction>
...
</commande>
```

Notre fichier XML de sociétés peut donc à présent s'écrire :

listeSociétés.xml (4)

```
<?xml version="1.0" encoding="UTF-8"?>

<!--Created by Vilvens on 19 avril 2002, 17:00 -->

<!DOCTYPE listeSociétés [
    <!ELEMENT listeSociétés ANY>
    <!ELEMENT société (nomSociété, adresse, contact, numéroEnregistrement)>
    <!ELEMENT nomSociété (#PCDATA)>
    <!ELEMENT adresse (rue, numéro, codePostal, ville)>
    <!ELEMENT rue (#PCDATA)>
    <!ELEMENT numéro (#PCDATA)>
    <!ELEMENT codePostal (#PCDATA)>
    <!ELEMENT ville (#PCDATA)>
    <!ELEMENT contact (nom, prénom, fonction, (telephone | email*))>
    <!ELEMENT nom (#PCDATA)>
    <!ELEMENT prénom (#PCDATA)>
    <!ELEMENT fonction (#PCDATA)>
    <!ELEMENT email (#PCDATA)>
    <!ELEMENT telephone (#PCDATA)>
    <!ELEMENT numéroEnregistrement (#PCDATA)>
    <!ATTLIST numéroEnregistrement ONSS ID #REQUIRED interne CDATA
            #IMPLIED>
]>
```

```
<listeSociétés>
  <société>
    <nomSociété>Genius S.A.</nomSociété>
    <adresse>
      <rue>rue de la Réussite</rue>
      <numéro>113</numéro>
      <codePostal>4500</codePostal>
      <ville>Visé</ville>
    </adresse>
    <contact>
      <nom>Bidendum</nom>
      <prénom>Albert</prénom>
      <fonction>General Manager</fonction>
      <email>abidend@groumf.com</email>
      <email>albert@sexy.net</email>
    </contact>
    <numéroEnregistrement ONSS="AG007" interne="BOND">500/213/40001
    </numéroEnregistrement>
  </société>
  <société>
    <nomSociété>La Truelle S.P.R.L.</nomSociété>
    <adresse>
      <rue>place de la construction</rue>
      <numéro>32</numéro>
      <codePostal>4000</codePostal>
      <ville>Liège</ville>
    </adresse>
    <contact>
      <nom>Pirandello</nom>
      <prénom>Luigi</prénom>
      <fonction>Contremaitre</fonction>
      <telephone>0497 337599</telephone>
    </contact>
    <numéroEnregistrement ONSS="SUR000" interne="Louche">500/220/40520
    </numéroEnregistrement>
  </société>
</listeSociétés>
```

A remarquer que si la deuxième société avait reçu comme attribut ONSS "AG007", lui aussi, le document aurait été déclaré invalide par le parser.

8. Les entités générales

8.1 Les entités textuelles

Un document XML peut en fait être beaucoup plus complexe parce qu'il peut manipuler, en plus des entités prédéfinies, des entités créées par l'utilisateur et connues par un identificateur. D'ailleurs, il est lui-même contenu dans une entité. Ces entités peuvent ensuite être référencées dans une autre partie du document. Le petit exemple suivant le fait voir clairement :

message.xml

```
<?xml version="1.0" encoding="UTF-8" ?>

<!—Created by Vilvens on 29 août 2002, 16:30 -->

<!DOCTYPE message [
    <!ELEMENT message ANY>
    <!ELEMENT importance EMPTY>
    <!ATTLIST importance notification CDATA #IMPLIED>
    <!ATTLIST importance niveau CDATA #IMPLIED>
    <!ELEMENT from (#PCDATA)>
    <!ELEMENT to (#PCDATA)>
    <!ELEMENT sent EMPTY>
    <!ATTLIST sent time CDATA #IMPLIED>
    <!ATTLIST sent date CDATA #IMPLIED>
    <!ELEMENT body (#PCDATA)>
    <!ENTITY to "<to>Students of INPRES-ISIL</to>">
    <!ENTITY luck "Good luck!">
]>

<message>
    <importance notification="loud_bell" niveau="high"/>
    <from>Prof Letueur</from>
    &to;
    <sent time="14:34" date="1999.01.24"/>
    <body>La présence au laboratoire est strictement obligatoire - grrr &luck;
    </body>
</message>
```

De telles entités sont encore appelées des "entités textuelles internes". Il existe également "entités textuelles externes" dont le texte de remplacement se trouve à une URI donnée.

8.2 Les entités paramètres

Il s'agit d'entités dont le texte n'est pas une donnée, mais un élément de définition XML général. Ainsi, si l'on définit :

```
<!ELEMENT client (nom, adresse, debit)>
<!ELEMENT courtier (nom, adresse, domaine)>
<!ELEMENT adversaire (compagnie, nom, adresse)>
```

on peut aussi écrire, pour éviter les répétitions et pouvoir apporter plus facilement des modifications :

```
<!ENTITY % identite "nom, adresse">
<!ENTITY % somme "debit">
<!ENTITY % champ "domaine">
<!ELEMENT client (%identite; , %somme;)gt;
<!ELEMENT courtier (%identite; , %champ;)gt;
<!ELEMENT adversaire (compagnie, %identite;)gt;
```

9. Les DTD externes

En pratique, un gros intérêt d'une DTD est de pouvoir **être partagée** par plusieurs fichiers XML, permettant ainsi **des échanges d'informations selon un véritable protocole de communication au niveau des données structurées**. Lorsque le DTD se trouve dans un fichier externe (dont l'extension est .dtd), cela devient possible, mais il faut alors utiliser une balise <!DOCTYPE> différente :

```
<!DOCTYPE nom_structure_informations (PUBLIC | SYSTEM) identification_DTD >
```

Deux approches sont donc possibles :

- ◆ soit on désigne le fichier .dtd explicitement, en précisant son URI de manière relative ou absolue derrière l'attribut SYSTEM :
- ◆ soit le DTD est officiellement enregistré auprès de l'ISO et il est alors désigné par un identifiant du type :

type (+ ou -) // propriétaire // description textuelle du DTD // langage

Reprendons notre exemple ci-dessus concernant des sociétés. Nous commençons par placer la DTD dans un fichier listeSocietes.dtd :

listeSocietes.dtd

```
<?xml encoding="UTF-8" ?>

<!-- Created by Vilvens on 26 mai 2002, 9:05 -->

<!ELEMENT listeSocietes ANY>
<!ELEMENT societe (nomSociété, adresse, contact, numéroEnregistrement)>
<!ELEMENT nomSociété (#PCDATA)>
<!ELEMENT adresse (rue, numéro, codePostal, ville)>
<!ELEMENT rue (#PCDATA)>
<!ELEMENT numéro (#PCDATA)>
<!ELEMENT codePostal (#PCDATA)>
<!ELEMENT ville (#PCDATA)>
<!ELEMENT contact (nom, prénom, fonction, (telephone | email*)) )>
<!ELEMENT nom (#PCDATA)>
<!ELEMENT prénom (#PCDATA)>
```

```
<!ELEMENT fonction (#PCDATA)>
<!ELEMENT email (#PCDATA)>
<!ELEMENT telephone (#PCDATA)>
<!ELEMENT numéroEnregistrement (#PCDATA)>
<!ATTLIST numéroEnregistrement ONSS ID #REQUIRED interne CDATA #IMPLIED>
```

Alors, un document XML faisant référence à cette DTD pourra être :

listeSociétés.xml (5)

```
<?xml version="1.0" encoding="UTF-8"?>

<!--Created by Vilvens on 19 avril 2002, 17:00 -->

<!DOCTYPE listeSociétés SYSTEM "D:\java-forte-
application\JMailSimplePart\listeSociétés.dtd"-->

<!DOCTYPE listeSocietes SYSTEM "listeSocietes.dtd">

<listeSocietes>
    <societe>
        <nomSociété>Genius S.A.</nomSociété>
        <adresse>
            <rue>rue de la Réussite</rue>
            <numéro>113</numéro>
            <codePostal>4500</codePostal>
            <ville>Visé</ville>
        </adresse>
        <contact>
            <nom>Bidendum</nom>
            <prénom>Albert</prénom>
            <fonction>General Manager</fonction>
            <email>abidend@groumf.com</email>
            <email>albert@sexy.net</email>
        </contact>
        <numéroEnregistrement ONSS="AG007" interne="BOND">500/213/40001
        </numéroEnregistrement>
    </societe>
    <societe>
        ...
    </societe>
</listeSocietes>
```

10. Les espaces de noms

A l'image des packages de Java ou des **namespaces** de C++, XML permet la définition d'espaces de noms, au sein desquels on peut définir des tags XML sans crainte de collisions avec d'autres tags extérieurs. L'objectif est donc bien clairement de permettre aux développeurs de créer des noms de balises sans devoir se soucier de l'existence éventuelle d'une balise de même nom dans un autre schéma d'information.

Dans notre fichier de sociétés, nous avons intuitivement contourné le problème de collision en utilisant la balise `<nom>` pour le contact, tandis que le nom de la société correspond à la balise `<nomSociété>` ;-) ... Mais nous avons sûrement pensé à l'époque à utiliser `<nom>` deux fois ! C'est possible si on utilise les espaces de noms – mais le prix à payer peut sembler élevé ...

Un espace de noms (namespace) se déclare dans une balise, selon la structure type :

```
<nom_balise xmlns:nom_namespace ="uri_en_rapport_avec_le_namespace">
...
</nom_balise>
```

L'URI a simplement pour rôle d'assurer que le namespace est unique et, donc, peut très bien désigner un site inexistant – en fait, on sert des URI pour assurer l'unicité. Les éléments qui seront définis à l'intérieur de la balise seront désignés dans la suite au moyen du **préfixe** formé par le nom du namespace. L'exemple simple suivant montre la syntaxe de déclaration :

```
<asbl xmlns:defi="http://www.inpres.be">
    <defi:livre>
        Programmation C++
    </defi:livre>
    <defi:auteur>
        Claude Vilvens
    </defi:auteur>
</asbl>
```

Dans cet exemple, defi est le nom du namespace et aussi le **préfixe**; livre est un nom **local** tandis que defi:livre est le **nom qualifié** ou **QName** ou **nom complet**.

A remarquer que la DTD (interne) éventuelle devra spécifier les préfixes de namespace définis. Ainsi, pour notre fichier de sociétés, cela donne :

listeSociétés.xml (avec namespaces)

```
<?xml version="1.0" encoding="UTF-8"?>

<!--Created by Vilvens on 19 may 2002, 18:00 --&gt;

&lt;!DOCTYPE listeSociétés [
    &lt;!ELEMENT listeSociétés ANY&gt;
    &lt;!ELEMENT société (s:nom, adresse, c:contact, numéroEnregistrement)&gt;
    &lt;!ELEMENT s:nom (#PCDATA)&gt;
    &lt;!ELEMENT adresse (rue, numéro, codePostal, ville)&gt;
    &lt;!ELEMENT rue (#PCDATA)&gt;
    &lt;!ELEMENT numéro (#PCDATA)&gt;
    &lt;!ELEMENT codePostal (#PCDATA)&gt;</pre>

```

```

<!ELEMENT ville (#PCDATA)>
<!ELEMENT c:contact (c:nom, c:prénom, c:fonction, (c:telephone | c:email*) )>
<!ELEMENT c:nom (#PCDATA)>
<!ELEMENT c:prénom (#PCDATA)>
<!ELEMENT c:fonction (#PCDATA)>
<!ELEMENT c:email (#PCDATA)>
<!ELEMENT c:telephone (#PCDATA)>
<!ELEMENT numéroEnregistrement (#PCDATA)>
<!ATTLIST numéroEnregistrement ONSS ID #REQUIRED interne CDATA
#IMPLIED>
]>

<listeSociétés xmlns:s="http://societe" xmlns:c="http://contact">
    <société>
        <s:nom>Genius S.A.</s:nom>
        <adresse>
            <rue>rue de la Réussite</rue>
            <numéro>113</numéro>
            <codePostal>4500</codePostal>
            <ville>Visé</ville>
        </adresse>
        <c:contact>
            <c:nom>Bidendum</c:nom>
            <c:prénom>Albert</c:prénom>
            <c:fonction>General Manager</c:fonction>
            <c:email>abidend@groumf.com</c:email>
            <c:email>albert@sexy.net</c:email>
        </c:contact>
        <numéroEnregistrement ONSS="AG007" interne="BOND">500/213/40001
        </numéroEnregistrement>
    </société>
    <société>
        <s:nom>La Truelle S.P.R.L.</s:nom>
        <adresse>
            <rue>place de la construction</rue>
            <numéro>32</numéro>
            <codePostal>4000</codePostal>
            <ville>Liège</ville>
        </adresse>
        <c:contact>
            <c:nom>Pirandello</c:nom>
            <c:prénom>Luigi</c:prénom>
            <c:fonction>Contremaître</c:fonction>
            <c:telephone>0497 337599</c:telephone >
        </c:contact>
        <numéroEnregistrement ONSS="SUR000" interne="Louche">500/220/40520
        </numéroEnregistrement>
    </société>
</listeSociétés>

```

11. Les schémas

11.1 Mieux que les DTDs

On peut reprocher aux DTDs, nonobstant leur relative simplicité, le fait qu'elles utilisent une syntaxe différente de XML et qu'elles de connaissent pas la notion d'espace de noms (avec les risques de collisions que ceci implique). De plus, les données n'y ont pas vraiment de type (elles sont représentées sous forme de chaînes de caractères) et il n'est pas possible de définir des spécifications ou des restrictions sur leurs valeurs.

Pour cette raison, le W3C a défini, à partir de 2000, la notion de schéma. Un schéma fait la même chose qu'une DTD, mais en mieux, c'est-à-dire que

- ◆ il utilise la notation XML;
- ◆ il connaît la notion de type de données;
- ◆ il gère les espaces de noms.

Un schéma est contenu dans un élément du genre:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">  
...  
</xsd:schema>
```

où xsd est un préfixe choisi arbitrairement et qui va permettre d'accéder, dans l'élément xsd:schema, à tous les symboles définis dans la syntaxe des schémas. Une fois le schéma sauvé dans un fichier d'extension xsd, tout document XML pourra y faire référence en déclarant, par exemple :

```
<listeSocietes ... xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance  
xsi:noNamespaceSchemaLocation="....xsd">  
</listeSocietes >
```

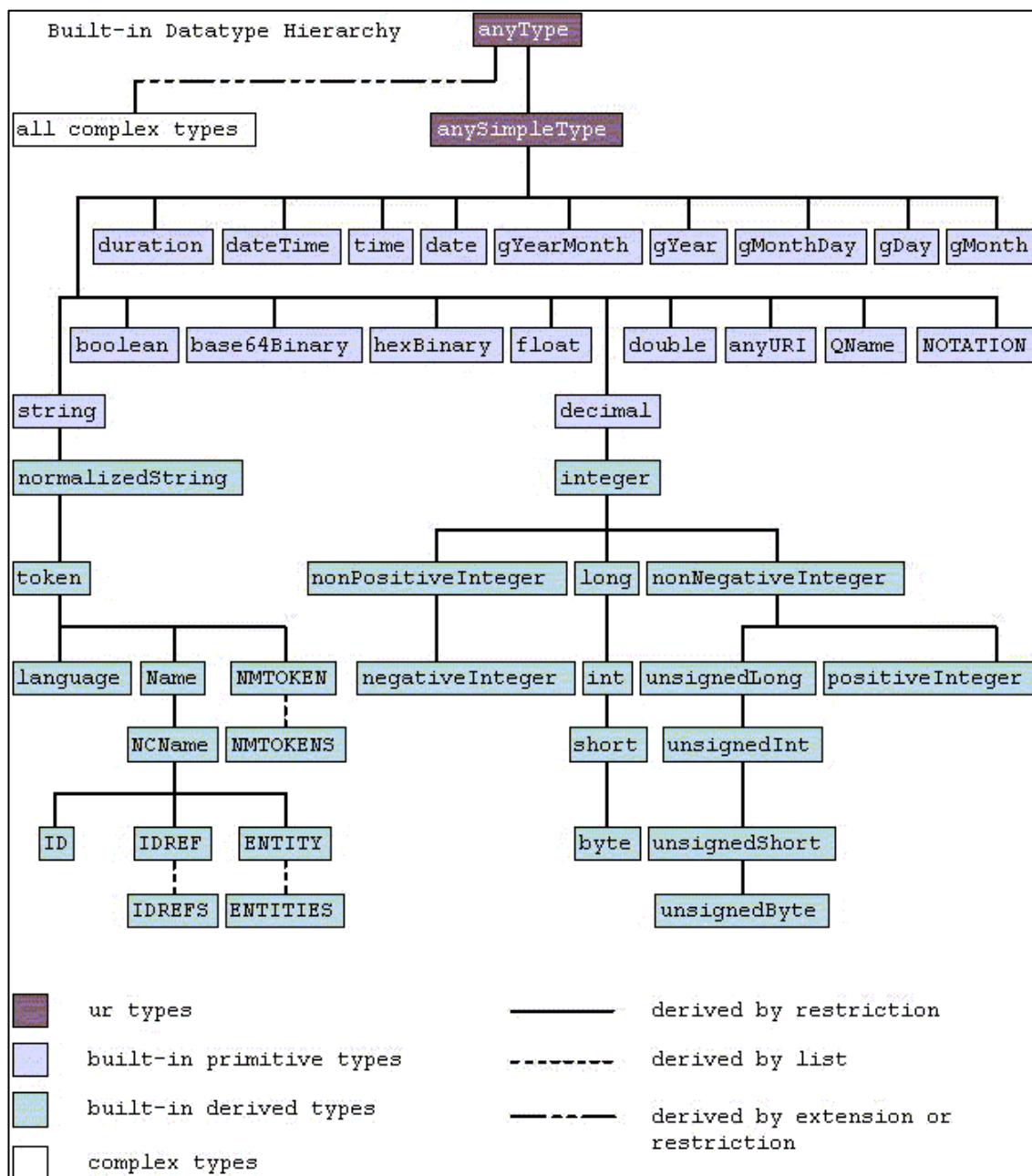
On parle encore de la création d'une **instance du schéma** (d'où le tag xsi). Voyons à présent ce que l'on peut placer entre les balises <xsd:schema> et </xsd:schema>.

11.2 La définition des éléments de type simple

Dans le contexte des schémas, on distingue les types de données qui sont des types simples (à valeur atomique sans attributs) des types complexes (qui sont soit structurés en sous-éléments, soit munis d'attributs, ou même les deux). Ceci est annoncé dans la définition d'un élément selon la syntaxe :

```
<xsd:element name="nombreEnfants" type="positiveInteger">.
```

où positiveInteger est un type prédéfini choisi parmi les suivants (les termes de la légende seront expliqués plus loin) :



On constate donc que l'on retrouve ici, notamment, les types de base des langage de programmation classique comme string, float ou integer. Le plus souvent, on définira ses propres types de données suivant la syntaxe :

<xsd:simpleType name=... > ... </xsd:simpleType>

ou

<xsd:complexType name=...> ... </xsd:complexType>

11.3 Les types dérivés des types simples

Les types simples définis par l'utilisateur sont construits à partir d'un type simple prédéfini, appelé le type de base, en précisant quelles valeurs sont possibles sont conservées. On parle alors de "types dérivés". Il existe en fait plusieurs techniques de dérivation.

1) la restriction

Parmi toutes les valeurs possibles du type de base, on n'en conserve que certaines. Pour spécifier ces valeurs, on utilise des propriétés du type de base appelées des "*facets*". Par exemple, pour le type **integer**, on dispose des facets suivants :

totalDigits, fractionDigits, pattern, whiteSpace, enumeration, maxInclusive, maxExclusive, minInclusive, minExclusive

Donc, on peut en user pour définir le type "JourDuMois" :

```
<xsd:simpleType name="JourDuMois">
  <xsd:restriction base="xsd:integer">
    <xsd:minInclusive value="1" />
    <xsd:maxInclusive value="31" />
  </xsd:restriction>
</xsd:simpleType>
```

Autre exemple, le type **string** dispose des facets

length, minLength, maxLength, pattern, enumeration, whiteSpace

et l'on peut ainsi définir le type "JourDeLaSemaine" de la manière suivante :

```
<xsd:simpleType name="JourDeLaSemaine">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="lundi" />
    <xsd:enumeration value="mardi" />
    <xsd:enumeration value="mercredi" />
    <xsd:enumeration value="jeudi" />
    <xsd:enumeration value="vendredi" />
    <xsd:enumeration value="samedi" />
    <xsd:enumeration value="dimanche" />
  </xsd:restriction>
</xsd:simpleType>
```

2) l'union

Le nouveau type inclut toutes les valeurs possibles d'un certain nombre d'autres types. Si on a au préalable défini le type "JourEnAbrege" par

```
<xsd:simpleType name="JourEnAbrege">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="lu" />
    <xsd:enumeration value="ma" />
    <xsd:enumeration value="me" />
    <xsd:enumeration value="je" />
    <xsd:enumeration value="ve" />
    <xsd:enumeration value="sa" />
    <xsd:enumeration value="di" />
  </xsd:restriction>
</xsd:simpleType>
```

on peut alors définir un type plus global "**JourSemaine**" par :

```
<xsd:simpleType name="JourSemaine">
    <xsd:union memberTypes="JourDeLaSemaine JourEnAbrege" />
</xsd:simpleType>
```

3) la liste

Ici, une valeur du nouveau type correspond à plusieurs valeurs du type de base séparées par un espace. Par exemple :

```
<xsd:simpleType name="JoursDOuverture">
    <xsd:list itemType="JrSemaine" />
</xsd:simpleType>
```

11.4 La définition des types complexes

Un type complexe peut contenir soit une définition de ses sous-éléments, soit une série d'attributs, soit les deux. Pour ce qui est des sous-éléments, on peut utiliser

1) la séquence

Les sous-éléments énumérés doivent être tous présents dans l'ordre indiqué; le nombre d'occurrences de chacun peut être déterminé par les attributs minOccurs et maxOccurs. Ainsi, le type

```
<xsd:complexType name="EnteteType">
    <xsd:sequence>
        <xsd:element name="titre" type="xsd:string" />
        <xsd:element name="auteur" type="xsd:string" />
        <xsd:element name="date" type="xsd:string" />
        <xsd:element name="lieu" type="xsd:string" />
    </xsd:sequence>
</xsd:complexType>
```

permet de déclarer un élément :

```
<xsd:element name="entete" type="EnteteType" />
```

avec dans le document XML les tags

```
<entete>
    <titre>...</titre>
    <auteur>...</auteur>
    <date>...</date>
    <lieu>...</lieu>
</entete>
```

2) le choix exclusif

Ici, un seul des sous-éléments énumérés doit être présent (exclusion mutuelle); le nombre d'occurrences de chaque alternative est déterminé par les attributs minOccurs et

maxOccurs de la déclaration de chaque sous-élément. Le nombre d'alternatives est déterminé par les attributs minOccurs et maxOccurs de la balise <xsd:choice>. Donc, par exemple :

```
<xsd:complexType name="EnfantsType">
    <xsd:choice minOccurs="0" maxOccurs="unbounded" >
        <xsd:element name="fils" type="PersonneType" />
        <xsd:element name="fille" type="PersonneType" />
    </xsd:choice>
</xsd:complexType>
```

3) l'enumération non ordonnée

Les sous-éléments énumérés doivent être tous présents au plus une fois, dans n'importe quel ordre; les attributs minOccurs et maxOccurs doivent valoir 0 ou 1. Donc :

```
<xsd:complexType name="NomType">
    <xsd:all>
        <xsd:element name="Prenom" type="xsd:string" />
        <xsd:element name="Nom" type="xsd:string" />
    </xsd:all>
</xsd:complexType>
```

4) l'extension

Ce mécanisme permet de réutiliser un type déjà défini, en lui ajoutant de nouvelles composantes (sous-éléments ou attributs). Ce premier exemple réalise une extension par attribut:

```
<xsd:complexType name="SommeDArgent">
    <xsd:simpleContent>
        <xsd:extension base="xsd:decimal">
            <xsd:attribute name="currency" type="xsd:string"/>
        </xsd:extension>
    </xsd:simpleContent>
</xsd:complexType>
```

tandis que celui-ci ajoute des composants :

```
<xsd:complexType name="ArticleType">
    <xsd:complexContent>
        <xsd:extension base="EnteteType">
            <xsd:sequence>
                <xsd:element name="VosReferences" type="xsd:string" />
                <xsd:element name="NosReferences" type="xsd:string" />
            </xsd:sequence>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

12. XSL et XSLT

12.1 Le principe

Une idée importante au sein de XML est de séparer les données de leur représentation. Il est en fait possible de visualiser les mêmes informations de différentes manières (on peut parler de création dynamique de documents, voire même de traduction dans un autre langage – comme HTML). Mieux encore, ces informations peuvent être utilisées de façons diverses pour obtenir de nouveaux fichiers XML : il s'agit alors de véritables transformations structurelles. Par exemple, on peut ainsi obtenir un fichier résultat au format Pdf, Postscript ou HTML.

Typiquement, l'application demandeuse d'une transformation est un browser et cette transformation consiste à générer le fichier HTML correspondant, dont ce browser saura effectivement se servir. Les problèmes d'incompatibilité entre browsers peuvent ainsi être contournés en créant plusieurs fichiers décrivant les transformations, chacun étant dédié à un type de browser particulier.

Les transformations seront effectuées par une application qui sera une application XSLT. C'est-à-dire ? **XSLT** (eXtensible Stylesheet Language Transformation) est un ensemble d'APIs permettant d'appliquer une transformation à un document XML. Un fichier écrit en **XSLT** est un fichier XML, mais d'extension **xsl**, qui décrit une transformation à appliquer à un fichier XML pour obtenir une nouvelle représentation c'est-à-dire un nouveau document utilisable par une application. Les documents XSL sont encore appelés des **feuilles de style** [*style sheet*]. Un processeur XSLT compare les éléments d'un document XML fourni en entrée aux modèles de la feuille de style. Quand il trouve un modèle adapté à l'élément en cours de traitement, il copie le résultat de l'application du modèle à l'élément dans l'arbre résultat qui sera fourni en sortie. Ce dernier peut éventuellement être sérialisé en un document XML ou autre.

12.2 La feuille de style selon XSLT

Un fichier XSL, mis à part son en-tête de fichier XML, comporte au début :

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">  
ou  
<xsl:transform version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
```

On constate donc l'existence d'un namespace **xsl** faisant référence à la norme de W3C. La transformation proprement dite est essentiellement décrite au moyen de balises

```
<xsl:template ...> </xsl:template>
```

On les appelle très logiquement en français des "modèles". Chaque transformation s'applique à un élément particulier désigné par l'attribut

```
match=<nom élément>"
```

En particulier, si le nom de l'élément est "/", la transformation s'applique à tout le document. On peut donc limiter un document xsl à ce seul modèle, si bien qu'un fichier xsl de base est du genre :

TransformSociété.xsl

```
<?xml version="1.0" encoding="UTF-8" ?>

<!--
Document : TransformSociété
Created on : 25 mai 2006
Author   : Vilvens
Comment
-->

<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:fo="http://www.w3.org/1999/XSL/Format">

<xsl:template match="/">
  ...
</xsl:template>

</xsl:stylesheet>
```

Mais on peut imaginer par exemple :

```
<xsl:template match="numéroEnregistrement">Numéro ONSS existant -- </xsl:template>
```

Ceci signifie : "chaque fois que l'on rencontre un tag 'numéroEnregistrement', écrire 'Numéro ONSS existant --' dans le fichier résultant". En pratique, on définit plutôt plusieurs templates en leur attribuant un nom, comme par exemple :

```
<xsl:template match="numéroEnregistrement" name="ONSSExists">Numéro ONSS
existant -- </xsl:template>
```

et on utilise le template correspondant au moyen de la balise :

```
<xsl:call-template name="ONSSExists"></xsl:call-template>
```

Mais pour exprimer de véritables transformations, nous avons besoin d'outils ...

12.3 Quelques balises xsl utiles

L'espace de noms xsl possède de nombreux tags. Nous allons nous contenter d'évoquer les plus utiles. Souvent, elles font référence à un nœud bien précis de l'arbre XML. En fait, pour sélectionner des éléments particuliers du document d'entrée, XSLT utilise des expressions rédigées en **XPath**. XPath est un langage non-XML utilisé pour identifier des parties particulières de documents XML – son nom se justifie donc pleinement. Notons simplement :

```
<xsl:value-of select="..."/>
```

Il s'agit ici de récupérer la valeur textuelle du nœud sélectionné pour l'insérer dans le document résultant.

<**xsl:text** disable-output-escaping="..."> ... </xsl:text>

On peut ainsi indiquer que le contenu de la balise doit être considéré comme un texte à ne pas interpréter. L'attribut disable-output-escaping, lorsqu'il est à "yes", permet d'insérer des caractères xml sans qu'ils soient interprétés (<, > et & pour "<", ">" et "&" respectivement).

<**xsl:if** test="(...)"> ... </xsl:if>

Ceci permet bien entendu d'insérer un élément si une condition (portant sur des éléments du fichier XML) est remplie. Bizarrement, il n'existe pas de clause "else" – en cas de nécessité, il faut utiliser la balise suivante :

```
<xsl:choose>
    <xsl:when test="....."> ....</xsl:when>
    <xsl:when test="....."> ....</xsl:when>
    ...
    <xsl:when test="....."> ....</xsl:when>
    <xsl:otherwise> ... </xsl:otherwise>
</xsl:choose>
```

<**xsl:for-each** select="..." ...> ... </xsl:for-each>

L'attribut select permet de désigner un nœud. L'instruction en elle-même est en fait une boucle : elle passe en revue tous les nœuds dont le nom est sélectionné pour appliquer le même traitement (le même modèle) à chacun. Il est possible d'obtenir un résultat trié en insérant après la balise ouvrante <xsl:for-each> la balise à corps vide <**xsl:sort** select="..." />.

<**xsl:output** .../>

Cette balise permet de déterminer le formatage du document produit. Il dispose de plusieurs attributs dont les deux plus utiles sont

- * method : "xml"|"html"|"txt"
- * encoding : le schème d'encodage

Par exemple :

```
<xsl:output method="html" encoding="ISO-8859-1" />
```

<**xsl:comment**>...</xsl:comment>

Ceci permet d'insérer un commentaire XML, donc de la forme <!-- ... -->.

Il existe également des fonctions XSLT que l'on peut utiliser dans la clause select de l'un des tags précisés ci-dessus. Citons, par exemple :

string-length(ch) : retourne la longueur de la chaîne ch;

substring-before (ch, tok) : extrait dans la chaîne ch ce qui précède la sous-chaîne tok;

substring-after (ch, tok) : extrait dans la chaîne ch ce qui suit la sous-chaîne tok;

translate (ch, txtrech, txtrempl) : dans ch, remplace txtrech par txtrech.

12.4 La transformation d'un fichier XML en page HTML

Pour fixer les idées, nous allons reprendre notre fichier listeSociétés.xml et générer une page HTML utilisant sa structure. Pour ce faire, nous allons créer le fichier xsl suivant :

société.xsl

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">

<xsl:template name="saut">
<xsl:text disable-output-escaping="yes">&lt;p&gt;</xsl:text>
</xsl:template>

<xsl:output method="html" encoding="ISO-8859-1" />

<xsl:template match="/">
<html>
<body>

<xsl:for-each select="listeSociétés/société">
Société associée : <h3><xsl:value-of select="nomSociété"/>*** </h3>
<xsl:text disable-output-escaping="yes">&lt;p&gt;</xsl:text>
Nom du contact : <b><xsl:value-of select="contact/nom"/></b>
<xsl:call-template name="saut"></xsl:call-template>

<xsl:for-each select="contact/email">
Son e-mail : <xsl:value-of select="."/>
</xsl:for-each>

<xsl:call-template name="saut"></xsl:call-template>

<xsl:for-each select="contact/telephone">
Son numéro de téléphone : <xsl:value-of select="."/>
</xsl:for-each>

<xsl:call-template name="saut"></xsl:call-template>
-----
<xsl:call-template name="saut"></xsl:call-template>
</xsl:for-each>

</body>
</html>
</xsl:template>

</xsl:stylesheet>
```

On remarquera en passant :

- ◆ le template "saut" qui insère un saut de ligne HTML, c'est-à-dire <p>;
 - ◆ les boucles for-each imbriquées.
-

Il nous faut à présent utiliser un processeur XSLT, par exemple **Xalan**, qui fait partie du projet Apache XML. Ce processeur

- ◆ reçoit en IN le fichier XML;
- ◆ reçoit en XSL le fichier XSL;
- ◆ produit en OUT un fichier que nous appellerons soc-transform.html.

Le code de la page HTML générée est :

```
soc-transform.html (fichier généré)
<html>
<body>
Soci&eacute;t&eacute; associ&eacute;e : <h3>Genius S.A.*** </h3><p>
Nom du contact : <b>Bidendum</b><p>
Son e-mail : abidend@groumf.com
Son e-mail : albert@sexy.net<p><p>
-----
<p>
Soci&eacute;t&eacute; associ&eacute;e : <h3>La Truelle S.P.R.L.*** </h3><p>
Nom du contact : <b>Pirandello</b><p><p>
Son num&eacute;ro de t&eacute;léphone : 0497 337599<p>
-----
<p>
</body>
</html>
```

c'est-à-dire qu'un browser quelconque saura afficher une telle page sans problème :

