

Java (II)
Programmation avancée des applications
classiques et cryptographie

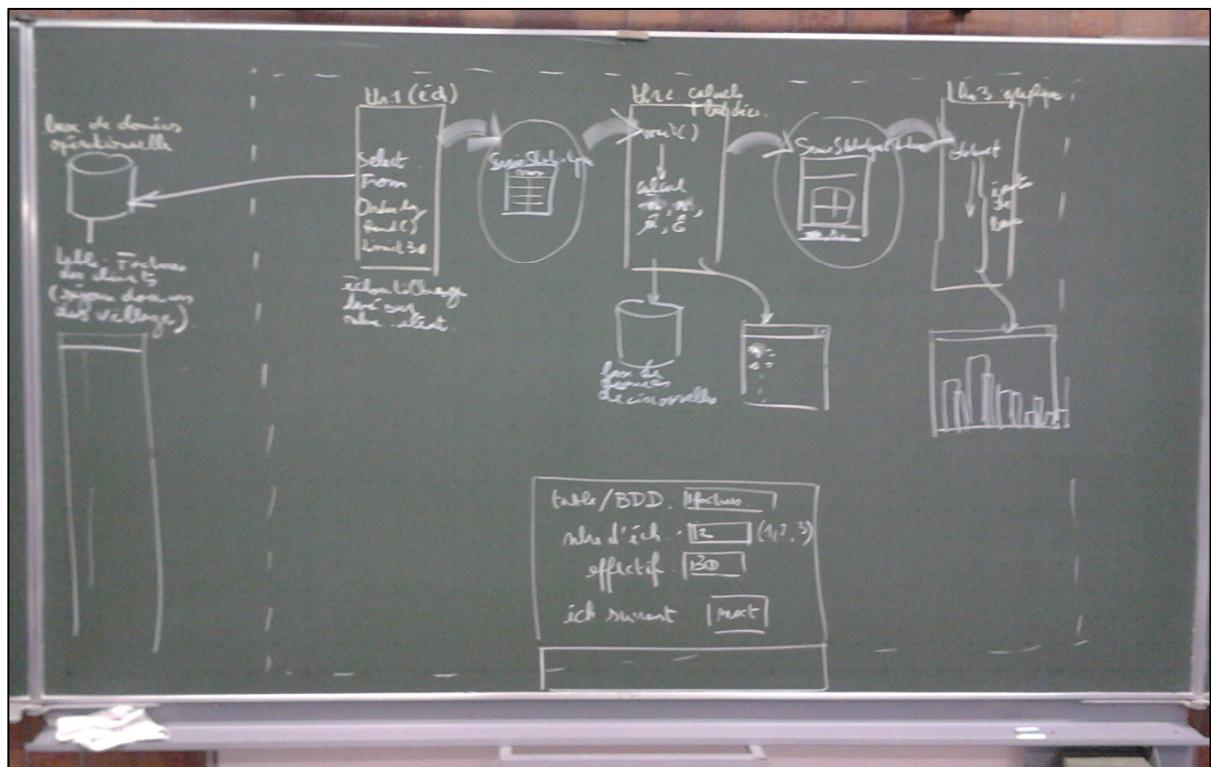
UE: Programmation réseaux, web et mobiles
AA : Réseaux et technologie Internet

Claude VILVENS

claude.vilvens@hepl.be

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

- I311 -
(2021-2022)



Confidentialité

$\text{long}(m) H(\text{univ} + \text{pwd} + "salt") = h \quad \text{univ} +$

$\hookrightarrow \text{"chiffre"} \quad \text{univ}$

$H(\text{univ} + \text{pwd} + \text{chiffre}) = h'$

$\text{échange} \rightarrow \text{gén. Ksh}$

$\text{de clés} \quad \left\{ \begin{array}{l} K_{sh} \\ K_{dh} \end{array} \right\} PK_{dh} = k$

$\Rightarrow DH \quad M_{dh}^x \quad A_{dh}^y$

$K_{dh} = M_{dh}^x A_{dh}^y$

$\text{confidentialité} \quad \{ msg \} PK_{dh} = y$

BDD

$h + "salt" \rightarrow H(\text{univ} + \text{pwd} + "salt") = h' \quad h = h'$

$\text{A univ} \rightarrow H(\text{univ} + \text{pwd} + \text{chiffre}) = h' \quad h = h'$

$\{ k \} P_n K_{dh} = K_{dh}$

$m_{dh} \rightarrow m_{dh}^x P_n$

$\{ B \cdot dh \cdot m_{dh}^x \} P_n = B \cdot dh \cdot m_{dh}^x = K_{dh}$

$\{ msg \} P_n K_{dh} = msg$

$\{ msg \} K_{dh} = msg$

$H(\{ msg \}) = k' \quad h = h' ?$

$h = msg + h \rightarrow H(\{ msg \} + msg \cdot K_{dh}) = h' \quad h = h' ?$

$= h \quad msg + h \rightarrow H(\{ msg \}) = h' \quad h = h' ?$

$msg + h \rightarrow \{ msg \} P_n K_{dh} = H(\{ msg \}) \cdot h \quad h = h' ?$

$H(\{ msg \}) = R \quad h = h' ?$

Sommaire



Introduction

X. Les threads

1. Définition	3
2. Une classe thread	3
3. Deux threads qui s'ignorent	6
4. Un autre manière de créer un thread	7
5. La synchronisation : les moniteurs	10
6. Les priorités	14
7. La synchronisation : l'attente d'un événement	15
8. Plus de deux threads	19
9. Les groupes de threads	20
10. La méthode stop() est deprecated	25
11. Une attente avec time-out	29
12. Le monitoring des threads sous Netbeans 5.5 ou 6.*	32
13. Les démons	33
14. La communication entre les threads : les "Pipes Streams"	
14.1 Un thread et un flux de sortie	33
14.2 Deux threads communicant par fichier	34
14.3 Deux threads communicant par tube (pipe)	36
15. Les threads prédefinis du framework AWT/Swing	38
16. Deux classes timers	
16.1 Deux classes timers	40
16.2 Le timer de java.util	40
16.3 Le timer de javax.swing	44
16.4 Une application GUI threadée : principe	45
16.5 Une application GUI threadée : implémentation	46
17. Les applets et les threads	
17.1 Une horloge qui ne s'affiche pas	51
17.2 Un thread dans une applet	53
18. Les problèmes d'animations	
18.1 Une animation simpliste	56
18.2 Eliminer les clignotements : ne pas effacer l'écran	56
18.3 Eliminer les clignotements : limiter la zone à reconstruire	57
18.4 Eliminer les clignotements : le double buffering	57
19. Les threads et les beans	
19.1 Un bean chronomètre	59
19.2 Le bean utilisateur : un générateur de nombres	61
19.3 Le container des deux beans	61

19.4 Sérialiser ce qui peut l'être	63
20. Un outil multithreadé pour Java Beans : la BeanBox	
20.1 Un outil de développement des beans	66
20.2 Placer un bean prédéfini dans la beanbox	67
21. Les synchroniseurs du JDK 1.5	
21.1 Les sémaphores	68
21.2 Les compteurs d'événements	75

XI. Les communications réseaux

1. Les protocoles d'Internet	80
2. Les ports	81
3. Une communication client-serveur basée TCP en mode console	
6.1 La classe Socket : le client	81
6.2 La classe ServerSocket : le serveur	84
4. Une communication client-serveur basée TCP en mode GUI	
4.1 Le client GUI	88
4.2 Le serveur GUI et son thread	90
5. Des caractères ou des bytes sur le réseau	
5.1 L'échange de caractères	94
5.2 L'échange de bytes	97
6. La communication par sérialisation	
6.1 Une information sérialisée	106
6.2 Le serveur d'objets sérialisés	107
6.3 Le client lecteur d'objets sérialisés	109
7. Un exemple simple de serveur multithread générique	
7.1 Le contexte et les objectifs	111
7.2 Un thread serveur et des threads clients	112
7.3 Quelques interfaces	112
7.4 Le thread serveur générique	115
7.5 Le thread client générique	117
7.6 L'implémentation de la source de tâches	118
7.7 L'application serveur	119
7.8 Les requêtes du protocole	120
7.9 L'application client	124
7.10 Le déploiement final du serveur et du client	126
8. Une communication réseau par datagrammes	
8.1 Des classes pour UDP	130
8.2 Une communication client-serveur par datagramme : le serveur	130
8.3 Une communication client-serveur par datagramme : le client	134
9. La communication réseau basée sur HTTP	
9.1 Les URLs	137
9.2 La classe URL	138
9.3 La classe URLConnection	140

XII. JDBC et l'accès aux bases de données

1. Présentation	143
2. Les modèles possibles	143
3. Etablir une connexion à une base de données	
3.1 Un contexte exemple	146
3.2 La classe Class et le chargement du pilote	147
3.3 La classe DriverManager	148
3.4 Obtenir une connexion	148
4. L'interface Connection	149
5. L'exécution d'une commande SQL	
5.1 Le siège de la commande SQL	150
5.2 Une requête de sélection et l'interface ResultSet	150
5.3 Les types SQL et les types Java	151
5.4 Une requête de mise à jour	152
6. Le programme complet	153
7. Un curseur plus souple	
7.1 Des ResultSet paramétrables	155
7.2 Un petit viewer de table	156
8. Les mises à jour au travers du curseur	
8.1 La modification	160
8.2 L'insertion	160
8.3 La suppression	160
8.4 Un viewer de table plus complet	161
9. Un peu de SQL dynamique	163
10. L'utilisation des dates et heures	167
11. Un autre exemple JDBC avec MySql	
11.1 Le serveur MySql	167
11.2 Utilisation des bases de données dans Netbeans	173
11.3 La portabilité des applications par rapport aux SGBDs	175
12. Les métadonnées	
12.1 Obtenir des informations sur une table	176
12.2 Une application : les requêtes scalaires	178
13. Erreurs, transactions, procédures stockées	179

XIII. La cryptographie et Java

1. Les caractéristiques idéales de la communication sécurisée	182
2. Les messages secrets : un petit historique	
2.1 Le codage de César et le principe de substitution	182
2.2 Les clés secrètes plus élaborées	184
2.3 Les cryptages de la Renaissance	184
2.4 Cryptage et codage	186
2.5 L'utilisation combinée de la substitution et de la transposition	187
2.6 Le cryptage par procédé électromécanique : Enigma	189
2.7 Le protocole de Diffie-Hellman : la clé partagée	191
2.8 La cryptographie à clé publique	192
3. Les bases des techniques cryptographiques	

3.1 Clés et algorithmes	193
3.2 Les modes de chiffrement	193
4. Les chiffrements symétriques	
4.1 Le principe général	194
4.2 Un exemple	194
4.3 Quelques algorithmes symétriques courants	195
4.4 Les générateurs de clés	196
5. Le protocole de Diffie-Hellman	
5.1 La base mathématique	196
5.2 L'algorithme	197
6. Les chiffrements asymétriques	
6.1 Le principe général	198
6.2 Un exemple	198
6.3 Quelques algorithmes asymétriques courants	199
6.4 Du bon usage des chiffrements asymétriques : les clés de session	200
7. Les classes de cryptographie : l'interface et l'implémentation	
7.1 L'interface de référence	201
7.2 Obtenir les classes : les méthodes factory	202
7.3 L'implémentation et les providers	202
7.4 Des providers complémentaires	203
7.5 Enregistrer un provider	205
8. Les limitations cryptographiques des JDK récents	209
9. Un cas pratique d'encryptage : un cryptage symétrique	
9.1 Un générateur de clé	212
9.2 La génération d'une clé	212
9.3 Obtenir un chiffrement	213
9.4 Le cryptage du message	213
9.5 Le décodage	214
9.6 Variante avec Rijndael : un chiffrement symétrique par blocs chaînés	215
9.6 L'envoi d'un message crypté par le réseau	217
10. Un deuxième cas pratique : un cryptage asymétrique	
10.1 L'obtention d'une paire de clés	222
10.2 Un cryptage-décryptage élémentaire	227
11. Les message digests	
11.1 Les fonctions de hachage	229
11.2 Un exemple	229
11.3 Quelques algorithmes de message digest courants	230
11.4 Utilisation des digests pour les mots de passe	230
11.5 Un exemple de login avec mot de passe	230
12. Les MACs et l'authentification légère	
12.1 Le principe du HMAC	236
12.2 La programmation des HMACs	237
13. Les signatures électroniques et l'authentification lourde	
13.1 Le principe de la signature électronique	243
13.2 La construction d'une signature basée sur un digest	243
13.3 Un exemple	244
13.4 Quelques algorithmes de signatures digitales courants	245
13.5 La signature d'un message transmis par le réseau	246

14. Les certificats	
14.1 Le principe	253
14.2 Vérifier un certificat	254
14.3 La gestion des clés	258
14.4 L'outil keytool	260
14.5 Créer un fichier certificat	261
14.6 Obtenir un certificat	262
14.7 Créer une entrée pour un certificat sûr	263
14.8 Les classes certificats en Java	264
15. L'outil Keytool IUI	
15.1 Présentation et installation	268
15.2 La création d'un keystore	269
15.3 La génération d'une paire de clés dans un keystore PKCS12	271
15.4 La génération d'une paire de clés dans un keystore JKS	274
15.5 L'exportation d'un certificat	274
15.6 L'importation d'un certificat	276
16. L'utilisation des keystores en programmation	
16.1 Afficher le contenu d'un keystore	279
16.2 Utilisation des deux keystores dans un processus de signature	281
17. Un protocole sécuritaire : SSL	
17.1 Une sécurisation au niveau des protocoles	285
17.2 Le protocoles SSL et HTTPS	285
17.3 Le dialogue SSL de création d'une session	286
17.4 Les sous-protocoles de SSL	286
18. Les jars signés	287
19. Le protocole Kerberos	
19.1 Le problème de l'authentification sans PKI	288
19.2 La version à 3 acteurs	289
19.3 Représentation schématique et notations	290
19.4 Réflexions et évolution	291
19.5 La version à 4 acteurs	292
19.6 Les sous-protocoles de Kerberos	293

XIV. Classes utiles et utilitaires : la suite

1. L'internationalisation	
1.1 Un objectif polyglotte	295
1.2 Les fichiers Properties	295
1.3 Le principe des bundles	296
1.4 Une boîte de dialogue de choix de langue	297
1.5 Une application internationale	299
2. Le Collections Framework	
2.1 La STL de Java	302
2.2 Les interfaces de base	302
2.3 Les interfaces plus spécifiques	303
2.4 Les classes de base "semi-abstraites"	304
2.5 Les classes containers courantes de type Collection	305

2.6 Un exemple élémentaire d'utilisation des collections	307
2.7 Les classes containers courantes de type Map	317
2.8 Les algorithmes classiques	317
2.9 Les évolutions génériques du JDK 1.5	320
3. Les impressions	
3.1 L'origine et le format des données à imprimer	322
3.2 Le service d'impression	323
3.3 Le job d'impression	324
3.4 Le programme de base	324
4. Les expressions régulières	
4.1 Le scalpel des chaînes de caractères	326
4.2 Les motifs	326
4.3 Les expressions régulières en Java	327
4.4 Un exemple de recherche et de test	328
5. Les compressions de fichiers	
5.1 Histoires de zip	331
5.2 Les sommes de contrôle	331
5.3 Les entrées d'un fichier zip	332
5.5 Un exemple console de création de fichier zip	334
5.6 Un GUI pour créer et visualiser les zips	337
6. JNI : les méthodes natives	
6.1 Aux frontières de la portabilité	344
6.2 L'application Java utilisatrice	345
6.3 Le fichier header pour la méthode native	347
6.4 L'implémentation en C/C++ de la méthode native dans une librairie partagée	348
6.5 L'exécution de l'application	350
7. Les fichiers de commandes Ant	351

XV. La librairie JFreeChart

1. Une bibliothèque pour graphiques statistiques	355
2. La représentation des données	356
3. L'exemple classique : le diagramme sectoriel	
3.1 Un Dataset particulier	357
3.2 Le JFreeChart correspondant	358
3.3 La classe factory	358
3.4 Les Plots	359
3.5 Le composant container visuel	359
3.6 L'application complète	360
3.7 La cascade des événements	361
3.8 Le diagramme de classes UML	363
4. L'utilisation d'une base de données	
4.1 Une classe facilitatrice	364
4.2 Les données dans une base de données	364
4.3 Le diagramme sectoriel à partir d'une base de données	365
5. D'autres graphiques classiques	

5.1 Les histogrammes comparés	25
5.2 Les graphiques linéaires d'évolution	29
6. Les graphiques de statistique à deux dimensions	
6.1 Le nuage de points	329
6.2 Les paramètres de régression et de corrélation	333
7. Les séries chronologiques	
7.1 Un axe des X avec des temps	380
7.2 Une graphique temporel à partir d'un base de données	383
8. Les graphiques dynamiques	
9. Les graphiques statistiques dans les applications Web	
9.1 Une applet d'affichage	389
9.2 Une servlet d'affichage	389
9.3 Un servlet fournisseur d'image	392
9.4 Une servlet fournisseur d'étude statistique	393

Annexe 1 : Les standards PKCS

Annexe 2 : La sérialisation et le serialVersionUID

Annexe 3 : Une introduction à JPA

1. Le couple Objets-Bases de données	405
2. L'architecture de JPA	406
3. Un exemple avec ObjectDB	408
4. La persistance des objets agrégés	411
5. Une unité de persistance MySQL	415

Annexe 4 : Les classes de certificats

Ouvrages consultés

Introduction



Dans un autre fascicule de notes de cours, il est écrit :

"Les présentes notes de cours sont le premier volume d'une série de cinq ouvrages tentant de donner une idée générale et relativement large de la programmation en Java. En simplifiant quelque peu, on pourrait dire que ce volume 1 vise à exposer la "programmation de base" en Java, tandis que le volume 2 visera les questions plus complexes (certains aiment parler de "programmation avancée" – pourquoi pas ;-) ...) et que le volume 3 présentera la "programmation des applications WEB". Les deux derniers volumes ne seront pas, à vrai dire, exclusivement consacrés à Java : en fait, ils parcourront des domaines informatiques très divers (des protocoles, des frameworks, des concepts d'architecture logicielle, ...) dont l'utilisation sera illustrée avec des librairies Java. Le volume 4 balayera ainsi la "programmation de protocoles applicatifs et de techniques de sécurité" tandis que le volume 5 abordera le "Java embarqué" et les "technologies d'e-commerce".

Donc, nous voilà au volume 2 ! Alors, quelles sont ces questions "plus complexes" ?

Eh bien, nous nous préoccupons tout d'abord de ce que nous avons à notre disposition dans d'autres langages comme C et C++ : les **threads** (incontournables dans une application Java professionnelle) et leur utilisation au sein des **Java Beans**, les **communications réseaux** (au niveau des protocoles TCP, UDP et HTTP primaire) et les accès aux **bases de données** : elles sont tellement pratiques que l'on ne voudrait plus gérer soi-même ses propres fichiers !

Arrivera alors ce qui sera sans doute ressenti comme le plus redoutable : l'arsenal des classes de **cryptographie** de Java. Clé secrète, clé publique, clé privée, digest, signature électronique, certificat, ... : il nous appartiendra de tout d'abord comprendre tous ces termes si courants dans le contexte du e-commerce et de les maîtriser, indépendamment du langage utilisé. Nous verrons ensuite comment utiliser ces concepts généraux dans notre programmation.

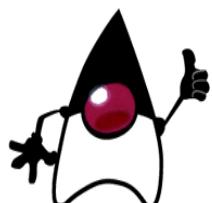
Un chapitre "fourre-tout" complètera enfin cette impression bien connue que l'on (presque) tout programmer en Java, impression renforcée par l'étude d'une librairie de graphiques statistiques bien pratique et orientée Java Beans ...

Avec un tel bagage, nous serons donc à même de concevoir et de développer une application Java classique, "classique" dans le sens "comme les applications C, C++ ou C#".

Mais il nous restera alors à nous attaquer aux développements WEBs en Java : il s'agit des fameuses servlets et des Java Server Pages. Pour cela, rendez-vous au volume III (**programmation des applications WEB**)

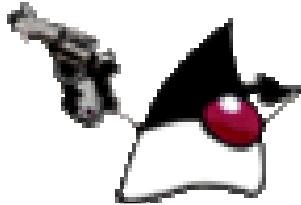
Claude Vilvens

P.S. Les exemples de programmation développés ici l'ont été avec NetBeans 8.*. La référence est le JDK 1.8.



Reprenez donc le fil de nos idées ! Les chapitres sont numérotés en poursuivant la séquence du volume I. Nous commençons donc par le chapitre X, qui va nous fournir l'un des éléments fondamentaux du développement Java ...

X. Les threads



L'appétit vient en mangeant, la soif s'en va en buvant.

(F. Rabelais, Gargantua)

1. Définition

Un thread est une "unité d'exécution", c'est-à-dire un flot d'instructions exécutables séquentiellement et qui ne peut être exécuté qu'au sein d'un programme. On qualifie encore les threads de "lightweight process" parce qu'ils fonctionnent au sein du contexte du programme qui les a lancés, par opposition à un **processus** qui travaille dans son propre environnement.

Les applets, que nous avons étudiées dans un chapitre précédent, font grand usage des threads. Une applet sera en effet d'autant mieux écrite qu'elle chargera un thread de réaliser une opération longue, n'empêchant ainsi pas l'applet de réaliser une autre tâche (finir d'être chargée, par exemple ...). Une autre utilisation classique des threads dans les applets est la réalisation de tâches répétitives propres aux animations.

Mais, bien sûr, un autre gros intérêt n'est pas de lancer un seul thread, mais bien plusieurs qui réalisent simultanément des tâches différentes. Ces threads s'exécuteront donc de manière **concurrente** : on parle encore **multithreading**.

2. Une classe thread

Les threads créés par un programmeur java peuvent hériter de la classe **Thread**, définie dans le package `java.lang`. Ses constructeurs réclament au préalable un nom pour le thread ou construisent ce nom d'après un compteur statique.

L'action réalisée par un thread est définie par le code de sa méthode **run()**, qui peut donc être redéfinie. Ce code comporte souvent une boucle si l'objectif est une animation, puisque l'on affichera les images successives.

Une autre façon de définir cette action est de créer un objet **Runnable** qui est ensuite passé au constructeur pour initialiser la variable membre **target**. La méthode **run()** du thread se contente alors d'invoquer celle de la variable membre **target** - nous en reparlerons notamment à propos des applets.

La classe Thread est définie, en première approche, comme suit :

classe Thread

```
public class Thread implements Runnable
{
    private char  name[];
    private int priority;
    ...
    private boolean      daemon = false;
    private boolean      stillborn = false;

    private Runnable target; /* What will be run. */
    ...
}
```

```
public final static int MIN_PRIORITY = 1;  
...  
public static native Thread currentThread();  
public static native void sleep(long millis) throws InterruptedException;  
  
public static void sleep(long millis, int nanos) throws InterruptedException {...}  
private void init(ThreadGroup g, Runnable target, String name){ ... }  
  
public Thread()  
{  
    init(null, null, "Thread-" + nextThreadNum());  
}  
public Thread(Runnable target)  
{  
    init(null, target, "Thread-" + nextThreadNum());  
}  
public Thread(ThreadGroup group, Runnable target)  
{  
    init(group, target, "Thread-" + nextThreadNum());  
}  
public Thread(String name)  
{  
    init(null, null, name);  
}  
public Thread(ThreadGroup group, String name)  
{  
    init(group, null, name);  
}  
public Thread(Runnable target, String name)  
{  
    init(null, target, name);  
}  
public Thread(ThreadGroup group, Runnable target, String name)  
{  
    init(group, target, name);  
}  
  
public synchronized native void start() {...}  
public void run()  
{  
    if (target != null)  
    {  
        target.run();  
    }  
}  
private void exit() {...}  
public final void stop()  
{  
    stop(new ThreadDeath());  
}
```

```

public final synchronized void stop(Throwable o)
{
    ...
}
public void interrupt()
{
    ...
}
public static boolean interrupted()
{
    return currentThread().isInterrupted(true);
}
public boolean isInterrupted()
{
    return isInterrupted(false);
}
public void destroy() {...}
public final native boolean isAlive() {...}

public final void join() throws InterruptedException { ... }

public final void suspend() {...}
public final void resume() {...}

public final void setPriority(int newPriority) {...}
public final int getPriority() {return priority;}
public final void setName(String name) {...}
public final String getName() {...}
public final ThreadGroup getThreadGroup() {return group;}
...
public final void setDaemon(boolean on) {...}
public final boolean isDaemon() {return daemon;}
...
public String toString() {...}

private native void setPriority0(int newPriority);
private native void stop0(Object o);
private native void suspend0();
private native void resume0();
private native void interrupt0();
}

```

On peut déjà remarquer :

- ◆ l'implémentation de *l'interface Runnable*, qui ne comporte que la méthode **void run()**;
- ◆ le polymorphisme forcené du **constructeur** : on peut y distinguer les versions utilisant un objet Runnable ou pas ainsi que les versions demandant un nom ou pas;
- ◆ la présence de méthodes dont l'effet, intuitivement, ne doit guère faire de doute : **init** (méthode privée), **start**, **stop**, **suspend**, **sleep**, etc;
- ◆ l'existence de la notion de priorité (variable membre *priority*);
- ◆ les méthodes privées *natives*, c'est-à-dire traitant directement avec le système d'exploitation de la machine hôte (elles sont le plus souvent écrites en C ou en C++); elles

sont inévitables ici puisque, en définitive, le thread fonctionne bel et bien sur cette machine hôte.

3. Deux threads qui s'ignorent

Considérons un premier exemple de lancement de deux threads simultanés. Le thread imaginé ici est très simple : il affiche un mot à intervalle de temps régulier, va à la ligne ou pas après chaque affichage et s'arrête quand il a atteint un nombre maximum d'affichages.

Il s'agit donc d'une simple classe qui réalise des affichages et le fait de vouloir faire exécuter ses affichages en parallèle avec d'autres actions revient simplement :

- à faire hériter la classe de la classe **Thread** de `java.lang`;
- à placer dans la méthode **run()** les affichages en question.

OuiNon.java

```
public class OuiNon extends Thread
{
    private String mot;
    boolean aLaLigne;
    private int pause;
    private int cptMax;

    public OuiNon (String m, int p, int c, boolean a)
    {
        mot = m; pause = p; cptMax = c; aLaLigne = a;
    }

    public void runtry
        {
            while (n < cptMax)
            {
                System.out.print(n + ". " + mot + " ");
                if (aLaLigne) System.out.print("\n");
                sleep(pause); n++;
            }
        }
        catch (InterruptedException e)
        {
            return;
        }
    }
}
```

La classe est vraiment minimale : un constructeur et la méthode `run()` redéfinie. On peut remarquer que l'exception lancée en cas de problème rencontré par cette dernière méthode doit être traitée d'une manière où l'autre (ici, on la traite en terminant le programme).

L'application va lancer deux threads : l'un dit "oui" pendant que l'autre dit "NON". La méthode de classe main() crée un objet thread instance de OuiNon et un objet temporaire OuiNon, tous deux immédiatement lancés par la méthode start() : celle-ci a pour effet que la Machine Virtuelle appelle la méthode run().

TestThreads.java

```
/*
 * TestThreads.java
 */
public class TestThreads
{
    public static void main(String args[])
    {
        OuiNon t = new OuiNon("Oui", 100, 20, true); t.start();
        new OuiNon("NON", 65, 45, false).start();
    }
}
```

On obtient à l'exécution :

```
0. Oui 0. NON
1. NON 2. NON 1. Oui
3. NON 4. NON 2. Oui
5. NON 3. Oui
6. NON 7. NON 4. Oui
8. NON 5. Oui
...
26. NON 27. NON 17. Oui
28. NON 18. Oui
29. NON 30. NON 19. Oui
31. NON 32. NON 33. NON 34. NON 35. NON 36. NON 37. NON 38. NON 39. NON 40.
NON 41. NON 42. NON 43. NON 44. NON
```

Belle cacophonie ...

4. Un autre manière de créer un thread

Dans le cas d'une classe qui hérite déjà d'une autre classe, il est impossible de la faire également hériter de la classe Thread pour permettre l'exécution en parallèle de l'une de ses méthodes. Ce serait le cas pour les filles de la classe suivante :

Afficheur.java

```
/*
 * Afficheur.java
 */
public class Afficheur
{
    protected String mot;
```

```

protected boolean aLaLigne;
protected int pause;
protected int cptMax;

public Afficheur (String m, int p, int c, boolean a)
{ mot = m; pause = p; cptMax = c; aLaLigne = a; }

public void affiche(String prompt)
{
    int n = 0;
    try
    {
        while (n<cptMax)
        {
            System.out.print(prompt + n + ". " + mot + " ");
            if (aLaLigne) System.out.print("\n");
            Thread.sleep(pause); n++;
        }
    }
    catch (InterruptedException e)
    {
        return;
    }
}
}
}

```

Dans ce cas, en fait assez fréquent lors d'un passage d'une application monothread à une version multithread, on considère que la classe fille va implémenter l'interface Runnable, qui ne comporte que la méthode

```
void run();
```

Cette méthode **run()** de Runnable est donc implémentée par la classe : en fait, la classe représente ainsi la fonction à faire exécuter par le thread. Cela pourra donner pour notre exemple :

NonOui.java

```

/*
 * NonOui.java
 */

public class NonOui extends Afficheur implements Runnable
{
    public NonOui (String m, int p, int c, boolean a)
    {
        super(m,p,c,a);
    }
}

```

```
public void run()
{
    System.out.println("J'affiche ..."); affiche("");
}
```

Le thread proprement dit sera instancié en utilisant le constructeur qui réclame comme premier argument l'objet qui implémente l'interface Runnable, soit ici un objet NonOui :

```
NonOui no1 = new NonOui("Oui", 100, 20, true);
Thread t1 = new Thread(no1); t1.start();
```

Le thread exécutera donc finalement, au sein de sa méthode run(), la méthode run() de la classe qui implémente Runnable ...

Notre cacophonie de threads peut donc s'écrire à présent comme suit :

TestThreads.java (2)

```
/*
 * TestThreads.java
 */

public class TestThreads
{
    public static void main(String args[])
    {
        NonOui no1 = new NonOui("Oui", 100, 20, true);    //no1.affiche("...");
        Thread t1 = new Thread(no1); t1.start();
        NonOui no2 = new NonOui("NON", 65, 45, false);   //no2.affiche("/**/");
        Thread t2 = new Thread(no2); t2.start();
    }
}
```

On obtient à l'exécution :

```
J'affiche ...
0. Oui
J'affiche ...
0. NON 1. NON 1. Oui
2. NON 3. NON 2. Oui
4. NON 3. Oui
5. NON 6. NON 4. Oui
...
25. NON 26. NON 17. Oui
27. NON 18. Oui
28. NON 29. NON 19. Oui
30. NON 31. NON 32. NON 33. NON 34. NON 35. NON 36. NON 37. NON 38. NON 39.
NON 40. NON 41. NON 42. NON 43. NON 44. NON
```

5. La synchronisation : les moniteurs

Le problème est bien connu de tout qui a déjà un peu travaillé dans un contexte de communication entre process : *certaines ressources ne peuvent être accédées simultanément* par plusieurs de ces process. Il en va de même pour les threads. On retrouve le problème connu de l'implémentation de **sections critiques**.

Dans ce contexte, Java permet de déclarer une méthode quelconque comme étant **synchronized** :

lorsqu'un thread invoque une telle méthode synchronized par l'intermédiaire d'un objet, celui-ci est **verrouillé** : un autre thread qui invoque une méthode synchronized (*la même ou une autre*) sur le même objet restera **bloqué** jusqu'à ce que le verrou soit enlevé par la fin de la première exécution de la méthode.

L'**exclusion mutuelle** est donc ainsi assurée. En fait, une méthode synchronized permet de définir, de manière transparente, ce que l'on appelle en théorie des systèmes d'exploitation, un "**monitor**".

Le terme "**monitor**", introduit par Hoare en 1974, désigne un ensemble de données et de procédures tel qu'**il ne peut y avoir, à un instant donné, qu'un seul processus ou thread qui y soit actif**, autrement dit qui se sert de l'une de ces procédures.

On dit encore que ce processus ou ce thread "*prend le monitor pour un objet donné*".

A titre d'exemple, considérons une classe "compte en banque" sur laquelle on peut effectuer un dépôt ou un retrait. Il est visiblement impératif que la prise de connaissance du compte et sa mise à jour constituent une **section critique**, c'est-à-dire une opération atomique que rien ne peut interrompre. La méthode de modification du compte sera donc synchronized. Le temps nécessaire pour effectuer l'opération permettra de briser l'effet de séquence trop régulier: afin de rompre cet effet de séquence, on place le thread en sommeil durant un temps aléatoire au moyen de la méthode déjà rencontrée

```
public static native void sleep(long millis) throws InterruptedException.
```

class compte

```
class compte
{
    protected double etat;
    protected String titulaire;

    public compte (double val, String nom) { etat = val; titulaire = nom; }

    public synchronized void modifCompte(double val, boolean depot, int numOp, int temps)
    /* dépôt (true) ou retrait (false) d'une somme val - numOp est le numéro d'opération */
    {
        System.out.println(numOp + ". " + "-- avant : " + etat );
        if (depot)
        {
            System.out.println("Depot ");
            etat += val;
        }
    }
}
```

```

        else
        {
            System.out.println("Retrait ");
            etat -= val;
        }
        try { Thread.sleep(temp); } catch (InterruptedException e) { };
        System.out.println(numOp + ". " + "-- après : " + etat);
    }
}

```

Un petit programme de test va nous convaincre de l'atomicité des transactions, comme dirait un prof' de bases de données :

MrCash.java

```

class compte { ...}

public class MrCash extends Thread
{
    private compte leCompte;
    private char operation; // R, D ou V
    private double somme;
    private int cptMax;
    private int pause;

    public MrCash (compte co, char op, double s, int c, int p)
    {
        leCompte = co; operation = op; somme = s; cptMax = c; pause = p;
    }

    public void run()
    {
        int n = 0;
        while (n<cptMax)
        {
            if (operation=='D') leCompte.modifCompte(somme,true,n, pause);
            else leCompte.modifCompte(somme,false,n, pause);
            n++;
        }
    }

    public static void main(String args[])
    {
        compte c = new compte(10000,"Vilvens");
        MrCash thrR = new MrCash(c, 'R', 6000, 3, 70);
        MrCash thrD = new MrCash(c, 'D', 2000, 5, 100);
        thrR.start();
        thrD.start();
    }
}

```

Un exemple d'exécution donne :

0. -- avant : 10000.0

Retrait

0. -- après : 4000.0

0. -- avant : 4000.0

Depot

0. -- après : 6000.0

1. -- avant : 6000.0

Depot

1. -- après : 8000.0

1. -- avant : 8000.0

Retrait

1. -- après : 2000.0

2. -- avant : 2000.0

Retrait

2. -- après : -4000.0

2. -- avant : -4000.0

Depot

2. -- après : -2000.0

3. -- avant : -2000.0

Depot

3. -- après : 0.0

4. -- avant : 0.0

Depot

4. -- après : 2000.0

A remarquer que le fait d'être endormi n'empêche pas le thread de rester en possession du moniteur !

Remarques

1) *Les moniteurs de Java sont réentrant*. Autrement dit, si un thread possède le moniteur, donc exécute une méthode synchronized, et que celle-ci invoque une autre méthode synchronized de la même classe, le moniteur acquis est conservé et la deuxième méthode peut être exécutée. Ainsi, si la classe compte possède une deuxième méthode getEtatCompte() synchronized :

class compte (2)

```
class compte
{
    protected double etat;
    protected String titulaire;

    public compte (double val, String nom) { etat = val; titulaire = nom; }

    public synchronized double getEtatCompte()
    {
        return etat;
    }
}
```

```

public synchronized void modifCompte(double val, boolean depot, int numOp, int temps)
/* dépôt (true) ou retrait (false) d'une somme val - numOp est le numéro d'opération */
{
    System.out.println(numOp + ". " + "-- avant : " + getEtatCompte());
    if (depot)
    {
        System.out.println("Dépot ");
        etat += val;
    }
    else
    {
        System.out.println("Retrait ");
        etat -= val;
    }
    try { Thread.sleep(temps); } catch (InterruptedException e) { };
    System.out.println(numOp + ". " + "-- après : " + getEtatCompte());
}
}

```

l'exécution donne un résultat identique à celui obtenu avec un accès direct à la variable membre état.

Une méthode comme `getEtatCompte()` se justifie en fait pleinement : Java ne fournit aucun moyen de synchroniser l'accès direct aux variables membres et tout se fait par l'intermédiaire des méthodes.

2) Une méthode statique peut aussi être `synchronized` : elle ne peut alors être exécutée que par un thread à la fois; le verrou implicite ainsi créé est sans effet sur les objets instanciant cette classe.

3) L'attribut `synchronized` n'est pas hérité : la redéfinition d'une méthode peut être `synchronized` ou pas; bien sûr, si elle appelle la méthode de la super-classe (`super...`), celle-ci posera un verrou sur l'objet durant son exécution.

4) Il est également possible de définir comme une section critique une suite d'instructions au lieu d'une méthode complète. Il suffit pour cela de placer cette suite dans un bloc ayant la syntaxe :

```

synchronized (<objet à verrouiller>)
{
    <instructions>
}

```

Tout se passera alors comme si les instructions constituaient une pseudo-méthode `synchronized` de l'objet utilisé. Une telle technique viole cependant les paradigmes de la P.O.O. et conduit à un code assez nébuleux.

6. Les priorités

Un thread fonctionne toujours à un certain niveau de priorité. Sans spécification précise, il tourne avec une priorité définie par **NORM_PRIORITY**, une constante définie dans le package.

Le scheduling utilisé ici est simple : les threads ayant les plus hautes priorités seront ceux qui se verront attribuer les différents processeurs de la machine hôte (dans le cas d'une machine mono-processeur, ce sera donc le thread le plus prioritaire qui sera exécuté). En cas de priorités égales, le choix se fait selon une file circulaire. Les threads de priorité moindre seront exécutés lorsque l'un des threads en cours d'exécution sera bloqué.

Il est toujours possible de modifier la priorité d'un thread, en restant dans les limites définies par deux autres constantes : **MIN_PRIORITY** et **MAX_PRIORITY**. On utilise pour cela la méthode :

```
void setPriority(int newPriority);
```

A priori, on donne une priorité basse aux threads traitant les affaires courantes et une priorité haute aux threads traitant des événements plus rares, comme une entrée d'utilisateur : on imagine en effet mal que l'application ne réagisse pas rapidement aux sollicitations de l'utilisateur en question.

Dans le cas de notre exemple, donnons une priorité maximale au thread de dépôt et une priorité minimale au thread de retrait :

MrCash.java (2)

```
....  
public static void main(String args[])
{
    compte c = new compte(10000,"Vilvens");
    MrCash thrR = new MrCash(c, 'R', 6000, 3, 70);
    MrCash thrD = new MrCash(c, 'D', 2000, 5, 100);
    thrD.setPriority(Thread.MAX_PRIORITY);
    thrR.setPriority(Thread.MIN_PRIORITY);
    thrR.start();
    thrD.start();
}
```

Le résultat est assez conforme à l'attente :

```
0. -- avant : 10000.0
Depot
0. -- après : 12000.0
1. -- avant : 12000.0
Depot
1. -- après : 14000.0
2. -- avant : 14000.0
Depot
2. -- après : 16000.0
3. -- avant : 16000.0
Depot
```

```
3. -- après : 18000.0
4. -- avant : 18000.0
Depot
4. -- après : 20000.0
0. -- avant : 20000.0
Retrait
0. -- après : 14000.0
1. -- avant : 14000.0
Retrait
1. -- après : 8000.0
2. -- avant : 8000.0
Retrait
2. -- après : 2000.0
```

On peut remarquer que bien que le thread de retrait ait été lancé le premier, c'est le thread de dépôt qui s'exécute d'abord. Cependant, il est dangereux de faire ce genre de supposition dans une situation quelconque : sur une autre plate-forme, le thread de retrait aurait peut-être eu l'occasion de s'exécuter une fois avant que le thread de dépôt ne prenne le pouvoir. Et que se passerait-il si la machine hôte était bi-processeur ? En fait, *l'exactitude d'un algorithme ne doit jamais dépendre de la priorité des threads impliqués.*

7. La synchronisation : l'attente d'un événement

La gestion des sections critiques n'est qu'un aspect des problèmes de synchronisation de processus ou de threads. Un autre aspect est le cas où un thread se met en attente de la réalisation d'un événement. L'exemple classique est celui du producteur-consommateur.

Un "producteur", disons ici un employé de banque, dépose un chèque dans le tiroir de transfert de son guichet. Un "consommateur", soit un mercenaire d'une compagnie de transport de fonds, soit un bandit en plein travail, retire ce chèque. Bien sûr, il ne peut retirer que

- ♦ si un chèque se trouve bien dans la zone de transfert;
- ♦ si l'opération de dépôt d'un chèque dans cette zone est entièrement terminée.

Ce dernier aspect des choses, à savoir l'atomicité d'une opération (ici, le dépôt – il en est de même du retrait d'ailleurs) est déjà pris en charge par la qualification "synchronized" que l'on donnera aux méthodes de dépôt et de retrait.

Pour ce qui est de l'attente sur un événement, à savoir "un chèque est disponible dans la zone de transfert", on utilisera **deux méthodes dérivées d'Object appelées obligatoirement au sein d'une méthode synchronized, donc au sein d'un monitor :**

- ♣ public final void **wait()**
Cette méthode place le thread en attente d'une notification par un autre thread. **Le moniteur est automatiquement libéré.**
- ♣ public final native void **notify()**
Cette méthode choisit un thread en attente sur le monitor et le réveille, **lui rendant automatiquement ce monitor.** Si il n'y a aucun thread en attente, la notification est perdue – elle n'est donc pas mémorisée.

Dans le cas de notre producteur-consommateur, on dispose en fait de trois acteurs :

- ♦ la zone de transfert, qui indique si elle est libre ou pleine et qui peut, de manière atomique, être remplie ou vidée :

ZoneTransfert.java

```
class ZoneTransfert
{
    private int contenu;
    private boolean occupé = false;

    public synchronized int retire() throws InterruptedException
    {
        while (occupé == false) wait();
        occupé = false;
        int sauvContenu = contenu;
        notify();
        return sauvContenu;
    }

    public synchronized void dépose(int somme) throws InterruptedException
    {
        while (occupé == true) wait();
        contenu = somme;
        occupé = true;
        notify();
    }
}
```

On pourrait être tenté de remplacer le **while** des méthodes `retire()` et `dépose()` par un **if** : mais ce serait supposer que la seule notification qui pourrait survenir serait due au fait que la condition est satisfaite, ce qui n'est pas forcément vrai dans un contexte de multithreading. Il faut donc que le thread, après être réveillé, teste encore cette condition, donc retourne dans sa boucle.

On remarquera encore que l'exception `InterruptedException`, dont il faut obligatoirement tenir compte, est simplement relancée.

- ♦ le producteur, qui place un chèque dans la zone de transfert quand c'est possible :

Producteur.java

```
class Producteur extends Thread
{
    private ZoneTransfert transit;
    private int nbreCheques;
    private String nomP;

    public Producteur (ZoneTransfert t, String n, int nc)
    {
        transit = t;
        nomP = n;
        nbreCheques = nc;
    }
}
```

```

public void run()
{
    for (int i=0; i< nbreCheques; i++)
    {
        int sommeDeposée = (int) (Math.random()*50000);
        try
        {
            transit.depose(sommeDeposée);
            System.out.println("Le producteur " + nomP + " dépose " + sommeDeposée
                +" Euros");
            int temps = (int) (Math.random()*2000);
            sleep(temps);
        }
        catch (InterruptedException e) {}
    }
}
}

```

Ici, il convient de remarquer plusieurs choses :

- ♣ la méthode **run()** appelle la méthode **depose()** de **ZoneTransfert**, qui lance l'exception **InterruptedException**; vu le prototypage de la méthode **run()**, il n'est pas possible de simplement la relancer et il faut donc la traiter (ce qui, dans notre cas, se résume à ne rien faire);
- ♣ la classe **Math**, définie dans le package par défaut **java.lang**, rassemble toutes les fonction habituelles de traitement des grandeurs numériques, comme **round()**, **ceil()**, **abs()**, **sin()**, etc. Inutile de dire que toutes ces méthodes sont statiques. Ici, pour générer le montant des chèques de manière aléatoire, nous utiliserons la méthode :

`public static synchronized double random()`

qui fournit un nombre décimal entre 0 et 1.

- ♦ le consommateur, qui retire un chèque de la zone quand c'est possible :

Consommateur.java

```

class Consommateur extends Thread
{
    private ZoneTransfert transit;
    private int nbreCheques;
    private String nomC;

    public Consommateur (ZoneTransfert t, String n, int nc)
    {
        transit = t;
        nomC = n;
        nbreCheques = nc;
    }
}

```

```
public void run()
{
    for (int i=0; i< nbreCheques; i++)
    {
        try
        {
            int sommeDeposée = transit.retire();
            System.out.println("Le consommateur " + nomC + " retire " + sommeDeposée
                               +" Euros");
        }
        catch (InterruptedException e) { }
    }
}
```

Il ne nous reste plus qu'à créer l'application qui lancera les threads :

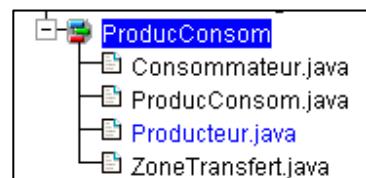
ProducConsom.java

```
public class ProducConsom
{
    public ProducConsom () {}

    public static void main(String args[])
    {
        ZoneTransfert t = new ZoneTransfert();
        Producteur p = new Producteur(t, "Batman", 5);
        Consommateur c = new Consommateur (t, "Superman", 5);

        p.start();
        c.start();
    }
}
```

Notre projet comporte donc en définitive 4 fichiers :



Le résultat de l'exécution sera :

```
Le producteur Batman dépose 47881 Euros
Le consommateur Superman retire 47881 Euros
Le producteur Batman dépose 841 Euros
Le consommateur Superman retire 841 Euros
Le producteur Batman dépose 7041 Euros
Le consommateur Superman retire 7041 Euros
Le producteur Batman dépose 22674 Euros
Le consommateur Superman retire 22674 Euros
```

Le producteur Batman dépose 37160 Euros
Le consommateur Superman retire 37160 Euros

Merveilleux synchronisme !

8. Plus de deux threads

On pourrait imaginer dans l'application précédente trois consommateurs pour un seul producteur :

ProducConsom.java (2)

```
public class ProducConsom
{
    public ProducConsom ()
    {
        public static void main(String args[])
        {
            ZoneTransfert t = new ZoneTransfert();
            Producteur p = new Producteur(t, "Batman", 15);
            Consommateur c1 = new Consommateur (t, "Superman", 5);
            Consommateur c2 = new Consommateur (t, "Bob Morane", 8);
            Consommateur c3 = new Consommateur (t, "Vilvens", 2);
            p.start();
            c1.start();
            c2.start();
            c3.start();
        }
    }
}
```

Il n'y aura évidemment aucun problème, *puisque le nombre de chèque déposés correspond au nombre total de chèques retirés*. Un exemple d'exécution sera :

Le producteur Batman dépose 4593 Euros
Le consommateur Vilvens retire 4593 Euros (chèque n°1)
Le producteur Batman dépose 23638 Euros
Le consommateur Superman retire 23638 Euros (chèque n°1)
Le producteur Batman dépose 32801 Euros
Le consommateur Bob Morane retire 32801 Euros (chèque n°1)
Le producteur Batman dépose 32105 Euros
Le consommateur Superman retire 32105 Euros (chèque n°2)
Le producteur Batman dépose 3931 Euros
Le consommateur Vilvens retire 3931 Euros (chèque n°2)
Le producteur Batman dépose 7392 Euros
Le consommateur Superman retire 7392 Euros (chèque n°3)
Le producteur Batman dépose 25748 Euros
Le consommateur Bob Morane retire 25748 Euros (chèque n°2)
Le producteur Batman dépose 425 Euros
Le consommateur Superman retire 425 Euros (chèque n°4)
Le producteur Batman dépose 44159 Euros

```
Le consommateur Bob Morane retire 44159 Euros (chèque n°3)
Le producteur Batman dépose 5016 Euros
Le consommateur Superman retire 5016 Euros (chèque n°5)
Le producteur Batman dépose 40205 Euros
Le consommateur Bob Morane retire 40205 Euros (chèque n°4)
Le producteur Batman dépose 29520 Euros
Le consommateur Bob Morane retire 29520 Euros (chèque n°5)
Le producteur Batman dépose 39541 Euros
Le consommateur Bob Morane retire 39541 Euros (chèque n°6)
Le producteur Batman dépose 31179 Euros
Le consommateur Bob Morane retire 31179 Euros (chèque n°7)
Le producteur Batman dépose 663 Euros
Le consommateur Bob Morane retire 663 Euros (chèque n°8)
```

On remarquera, encore une fois, qu'il serait dangereux de poser la moindre hypothèse quant à l'ordre d'exécution ...

Remarque

On obtiendrait le même résultat si on avait utilisé dans la méthode `depose()` la méthode
`public final native void notifyAll()`

qui réveille tous les threads en attente du moniteur. De toute façon, un seul parviendra à s'en saisir ...

9. Les groupes de threads

Et si le nombre de chèque déposés est inférieur au nombre total de chèques que les consommateurs souhaitent retirer ?

Le problème est évident : certains threads resteront éternellement en attente ☹ ... Le producteur peut tuer les threads restant en usant de la notion de groupe de threads.

Dans un contexte de multithreading, il est fréquent de répartir les différents threads dans des groupes. La raison principale de cette façon d'agir est la sécurité : en effet, un thread ne peut modifier le comportement d'un thread appartenant à un autre groupe. De plus, comme un groupe peut contenir un autre groupe, on peut aboutir à la création d'une hiérarchie de groupes de threads.

Un thread appartient toujours à un groupe, fût-ce celui par défaut qui s'appelle très finement "**main**". Ce groupe est une instance de la classe **ThreadGroup**, dérivée d'Object et qui ne possède que deux constructeurs :

```
public ThreadGroup(String name);
public ThreadGroup(ThreadGroup parent, String name);
```

Un groupe de threads doit donc toujours avoir un nom; on peut éventuellement préciser, lors de sa création, le groupe parent. Cette classe propose quelques méthodes bien pratiques; citons :

- ◆ public final String **getName()**
On obtient évidemment le nom du groupe.

◆ **public int activeCount()**

Elle fournit une estimation du nombre de threads actifs dans le groupe – une estimation puisqu'il se peut que des threads se soient terminés ou au contraire aient été créés entre le moment de l'évaluation et le moment de la prise de connaissance de la valeur.

◆ **public int enumerate(Thread list[])**

Cette méthode remplit un tableau de Threads avec tous les threads actifs dans le groupe et ses sous-groupes.

Nous allons réécrire le producteur de notre problème en lui demandant d'afficher en fin de traitement le nom des threads encore actifs et de les tuer (sauf, évidemment, lui-même et le thread principal, qui s'appelle lui aussi **main**). Pour cela il nous faut acquérir l'objet groupe du thread courant : celui-ci s'obtient avec la méthode de classe

public static native Thread currentThread()

et la méthode

public final ThreadGroup getThreadGroup()

De plus, nous donnons à nos threads des noms plus expressifs en appelant, au sein de leur constructeur, le constructeur de la classe mère avec un paramètre nom. Cela donne :

Producteur.java (2)

```
class Producteur extends Thread
{
    private ZoneTransfert transit;
    private int nbreCheques;
    private String nomP;

    public Producteur (ZoneTransfert t, String n, int nc)
    {
        super(n); transit = t; nomP = n; nbreCheques = nc;
    }

    public void run()
    {
        System.out.println("Je suis le producteur et le thread : " + getName());
        for (int i=0; i< nbreCheques; i++)
        {
            int sommeDeposée = (int) (Math.random()*50000);
            try
            {
                transit.depose(sommeDeposée);
                System.out.println("Le producteur " + nomP + " dépose " + sommeDeposée
                    +" Euros");
                int temps = (int) (Math.random()*2000); sleep(temps);
            }
            catch (InterruptedException e) {}
        }
    }
}
```

```

System.out.println("Il n'y a plus de chèques !!!");
System.out.println("Arret des autres threads !!!");
Thread[] listeDesThreads;
ThreadGroup groupeCourant = Thread.currentThread().getThreadGroup();
int nbreThreads = groupeCourant.activeCount();
System.out.println("Il y a " + nbreThreads + " threads encore actifs");
listeDesThreads = new Thread[nbreThreads];
groupeCourant.enumerate(listeDesThreads);
for (int j=0; j<nbreThreads; j++)
{
    String nomT = listeDesThreads[j].getName();
    System.out.println("Thread n° " + j + " = " + nomT);

    if (nomT.equals ("Distributeur") == false && nomT.equals ("main") == false)
    {
        listeDesThreads[j].stop();
    }
}
System.out.println("Fin Arret des autres threads !!!");
System.out.println("Reste-t-il des threads ?"); // Vérification

try
{
    sleep(3000); // le temps que les threads tués disparaissent
}

catch (InterruptedException e) {}

Thread[] listeDesThreads2;
ThreadGroup groupeCourant2 = Thread.currentThread().getThreadGroup();
int nbreThreads2 = groupeCourant2.activeCount();
System.out.println("Il reste " + nbreThreads2 + " threads actifs");
listeDesThreads2 = new Thread[nbreThreads2];
groupeCourant2.enumerate(listeDesThreads2);
for (int j=0; j<nbreThreads2; j++)
{
    String nomT = listeDesThreads2[j].getName();
    System.out.println("Thread n° " + j + " = " + nomT);
}
System.out.println("Le producteur a fini !!!");
}
}

```

Le consommateur, mis à part le fait qu'il porte un nom, affichera aussi un message s'il est parvenu à mener sa tâche :

Consommateur.java (2)

```
class Consommateur extends Thread
{
    private ZoneTransfert transit;
    private int nbreCheques;
    private String nomC;

    public Consommateur (ZoneTransfert t, String n, int nc)
    {
        super(n);
        transit = t;
        nomC = n;
        nbreCheques = nc;
    }

    public void run()
    {
        for (int i=0; i< nbreCheques; i++)
        {
            try
            {
                int sommeDeposée = transit.retire();
                System.out.println("Le consommateur " + nomC + " retire " + sommeDeposée +
                    " Euros (chèque n°" + (i+1) + ")");
            }
            catch (InterruptedException e) {}
        }
        System.out.println("Le consommateur " + nomC + " est parvenu à terminer son travail");
    }
}
```

L'application lanceuse de threads va donc créer une situation à problème : pas assez de chèques pour tout le monde :

ProducConsom.java (3)

```
public class ProducConsom
{
    public ProducConsom () { }

    public static void main(String args[])
    {
        ZoneTransfert t = new ZoneTransfert();
        Producteur p = new Producteur(t, "Distributeur", 7);
        Consommateur c1 = new Consommateur (t, "Superman", 5);
        Consommateur c2 = new Consommateur (t, "Bob Morane", 8);
        Consommateur c3 = new Consommateur (t, "Vilvens", 2);
        // gloups ! 7 < 5+8+2 !!!
    }
}
```

```
p.start();
c1.start();
c2.start();
c3.start();

        System.out.println("Les threads sont lancés !!!");
    }
}
```

Un exemple d'exécution est :

Les threads sont lancés !!!

Je suis le producteur et le thread : Distributeur

>>>> depot n° 1

Le producteur Distributeur dépose 38893 Euros

Le consommateur Superman retire 38893 Euros (chèque n°1)

>>>> depot n° 2

Le producteur Distributeur dépose 37189 Euros

Le consommateur Vilvens retire 37189 Euros (chèque n°1)

>>>> depot n° 3

Le producteur Distributeur dépose 48996 Euros

Le consommateur Superman retire 48996 Euros (chèque n°2)

>>>> depot n° 4

Le producteur Distributeur dépose 21506 Euros

Le consommateur Bob Morane retire 21506 Euros (chèque n°1)

>>>> depot n° 5

Le producteur Distributeur dépose 28181 Euros

Le consommateur Vilvens retire 28181 Euros (chèque n°2)

Le consommateur Vilvens est parvenu à terminer son travail

>>>> depot n° 6

Le producteur Distributeur dépose 32831 Euros

Le consommateur Superman retire 32831 Euros (chèque n°3)

>>>> depot n° 7

Le producteur Distributeur dépose 6231 Euros

Le consommateur Bob Morane retire 6231 Euros (chèque n°2)

Il n'y a plus de chèques !!!

Arret des autres threads !!!

Il y a 4 threads encore actifs

Thread n° 0 = main

Thread n° 1 = Distributeur

Thread n° 2 = Superman

Thread n° 3 = Bob Morane

Fin Arret des autres threads !!!

Reste-t-il des threads ?

Il reste 2 threads actifs

Thread n° 0 = main

Thread n° 1 = Distributeur

Le producteur a fini !!!

Tous les threads sont donc bien arrêtés définitivement. Il convient cependant de remarquer que la méthode stop() doit être utilisée avec circonspection, puisqu'elle "tue" le thread() quelle que soit son activité et quel que soit l'état des données qu'il manipule. Ce qui explique que ...

Remarque

Dans le cas où l'application ci-dessus est développée et exécutée sous Netbeans, il convient plutôt de cibler les threads à arrêter plutôt que ceux à conserver en vie : en effet, l'EDI prévoit le lancement de threads complémentaires destinés à optimiser la gestion d'une application GUI.

10. La méthode stop() est deprecated

Une consultation de l'aide du JDK nous indique effectivement :

```
public final void stop()
```

Deprecated. ...

La justification a déjà été évoquée ci-dessus : un thread victime de stop() libère tous les moniteurs qu'il avait pris, avec comme conséquence éventuelle et même probable de laisser ces objets dans des états inconsistants (puisque les sections critiques ont été abandonnées avant la fin des opérations nécessairement atomiques). On parle encore de *damaged objects* pour qualifier ces objets que l'on peut encore considérer comme "corrompus".

Bien sûr, la question fuse aussitôt : ***comment faut-il alors procéder pour contraindre de manière sûre un thread à s'arrêter ?*** Le plus rationnel est d'utiliser une variable membre indiquant l'état attendu du thread : celui-ci vérifiera alors l'état de cette variable pour en déduire éventuellement qu'il doit arrêter son exécution. Une sage précaution, si on se limite à la consultation de cette variable d'état, sera de la déclarer comme étant "**volatile**" ou de prévoir son accès au sein d'un moniteur.

En particulier, on peut soumettre l'action du thread à un test répétitif de sa référence : tant qu'elle désigne le thread courant, le thread poursuit son action. Celle-ci s'interrompt quand on met cette référence à null (nous en reparlerons au sujet des threads utilisés dans les applets).

Evidemment, si le thread est bloqué sur un wait(), le test d'une variable d'état ne peut se faire. Dans ce cas, on utilise au préalable la méthode :

```
public void interrupt()
```

qui met fin aux attentes induites par un appel de wait(), join() ou sleep() en lançant une exception instance de **InterruptedException**.

Si nous reprenons notre programme de producteur-consommateurs de chèques en utilisant cette dernière méthode, les modifications seront les suivants :

a) pour le consommateur : une variable membre booléenne nommée "interrompu" a été ajoutée; initialement à false, elle passe à true si l'exception **InterruptedException** a été captée; de plus, les consommateurs feront partie d'un groupe de threads (qui comprendra aussi le producteur), ce qui évitera d'éventuelles confusions avec d'autres threads (comme par exemple les threads système des applications GUIs Java ;-) – voir plus loin).

Consommateur.java (3)

```

class Consommateur extends Thread
{
    private ZoneTransfert transit;
    private int nbreCheques;
    private String nomC;

    private volatile boolean interrompu = true;

    public Consommateur (ZoneTransfert t, ThreadGroup tg, String n, int nc)
    {
        super(tg, n);
        transit = t;
        nomC = n;
        nbreCheques = nc;
    }

    public void run()
    {
        interrompu = false;
        for (int i=0; i< nbreCheques && !interrompu; i++)
        {
            try
            {
                int sommeDeposée = transit.retire();
                System.out.println("Le consommateur " + nomC + " retire " + sommeDeposée
                    + " Euros (chèque n°" + (i+1) + ")");
            }
            catch (InterruptedException e)
            {
                System.out.println(getName() + "> Interruption reçue");
                interrompu = true;
                System.out.println("Le consommateur " + nomC + " a été contraint de s'arrêter");
                // break; -- pour un for simple - Denys-like ;-
            }
        }
        if (!interrompu)
            System.out.println("Le consommateur " + nomC +
                " est parvenu à terminer son travail");
    }
}

```

b) pour le producteur : il appartient au même groupe que ses consommateurs et avise ceux-ci de la fin de la production de chèques en appelant leur méthode interrupt(), ce qui les amènera à se terminer proprement.

Producteur.java (3)

```

class Producteur extends Thread
{
    private ZoneTransfert transit;
    private int nbreCheques;
    private String nomP;

    public Producteur (ZoneTransfert t, ThreadGroup tg, String n, int nc)
    {
        super(tg, n);
        transit = t;
        nomP = n;
        nbreCheques = nc;
    }

    public void run()
    {
        System.out.println("Je suis le producteur et le thread : " + getName());

        for (int i=0; i< nbreCheques; i++)
        {
            int sommeDeposée = (int) (Math.random()*50000);
            try
            {
                transit.depose(sommeDeposée);
                System.out.println("Le producteur " + nomP + " dépose " + sommeDeposée
                    +" Euros");
                int temps = (int) (Math.random()*2000); sleep(temps);
            }
            catch (InterruptedException e) { }
        }
        System.out.println("Il n'y a plus de chèques !!!!");
        System.out.println("Arret des autres threads !!!!");
        Thread[] listeDesThreads;
        ThreadGroup groupeCourant = Thread.currentThread().getThreadGroup();
        int nbreThreads = groupeCourant.activeCount();
        System.out.println("Il y a " + nbreThreads + " threads encore actifs dans le groupe "
        + groupeCourant.getName());
        listeDesThreads = new Thread[nbreThreads];
        groupeCourant.enumerate(listeDesThreads);
        for (int j=0; j<nbreThreads; j++)
        {
            String nomT = listeDesThreads[j].getName();
            System.out.println("Thread n° " + j + " = " + nomT);
            if (nomT.equals ("Distributeur") == false)
            {
                listeDesThreads[j].interrupt();
            }
        }
    }
}

```

```
System.out.println("Fin des autres threads !!!");  
  
System.out.println("Reste-t-il des threads ?"); // Vérification – rien de neuf  
...  
System.out.println("Le producteur a fini !!!");  
}  
}
```

c) pour l'application mère des threads : elle instancie un objet groupe de threads et y place tous les acteurs.

ProducConsom.java (4)

```
public class ProducConsom  
{  
    public ProducConsom () {}  
    public static void main(String args[])  
    {  
        ZoneTransfert t = new ZoneTransfert();  
        ThreadGroup groupeCheques = new ThreadGroup ("CHEQUES");  
        Producteur p = new Producteur(t, groupeCheques, "Distributeur", 7);  
        Consommateur c1 = new Consommateur (t, groupeCheques, "Superman", 5);  
        Consommateur c2 = new Consommateur (t, groupeCheques, "Bob Morane", 8);  
        Consommateur c3 = new Consommateur (t, groupeCheques, "Vilvens", 2);  
        // gloups ! 7 < 5+8+2 !!!  
  
        p.start();  
        c1.start();    c2.start();    c3.start();  
        System.out.println("Les threads sont lancés !!!");  
    }  
}
```

L'exécution de notre application remodelée n'est guère différente de celle de la précédente version :

Je suis le producteur et le thread : Distributeur

```
Le producteur Distributeur dépose 42186 Euros  
Le consommateur Superman retire 42186 Euros (chèque n°1)  
Les threads sont lancés !!!  
Le consommateur Superman retire 37248 Euros (chèque n°2)  
Le producteur Distributeur dépose 37248 Euros  
Le consommateur Vilvens retire 47313 Euros (chèque n°1)  
Le producteur Distributeur dépose 47313 Euros  
Le consommateur Bob Morane retire 8851 Euros (chèque n°1)  
Le producteur Distributeur dépose 8851 Euros  
Le producteur Distributeur dépose 14398 Euros  
Le consommateur Superman retire 14398 Euros (chèque n°3)  
Le producteur Distributeur dépose 4588 Euros  
Le consommateur Vilvens retire 4588 Euros (chèque n°2)  
Le consommateur Vilvens est parvenu à terminer son travail
```

Le consommateur Bob Morane retire 16213 Euros (chèque n°2)

Le producteur Distributeur dépose 16213 Euros

Il n'y a plus de chèques !!!!

Arret des autres threads !!!!

Il y a 3 threads encore actifs dans le groupe CHEQUES

Thread n° 0 = Distributeur

Thread n° 1 = Superman

Superman> Interruption reçue

Le consommateur Superman a été contraint de s'arrêter

Thread n° 2 = Bob Morane

Bob Morane> Interruption reçue

Le consommateur Bob Morane a été contraint de s'arrêter

Fin des autres threads !!!!

Reste-t-il des threads ?

Il reste 1 threads actifs

Thread n° 0 = Distributeur

Le producteur a fini !!!!

11. Une attente avec time-out

Une autre alternative à l'évitement de stop(), dans le cas de notre application à chèques, est d'imaginer que les consommateurs testent périodiquement le fait que le producteur est toujours en activité et s'arrêtent dès qu'ils constatent sa disparition. On pourrait penser pour cela à utiliser la méthode :

```
public final native boolean isAlive()
```

mais il n'est pas immédiat de disposer d'une référence du thread visé. Nous allons donc procéder plus simplement en utilisant un flag public (appelons-le "vivant") qui sera testé périodiquement.

Producteur.java (3)

```
class Producteur extends Thread
{
    private ZoneTransfert transit;
    private int nbreCheques;
    private String nomP;
    public static boolean vivant = true;

    public Producteur (ZoneTransfert t, String n, int nc) { ... }
    public void run()
    {
        System.out.println("Je suis le producteur et le thread : " + getName());
        ...
        System.out.println("Le producteur a fini !!!!");
        vivant = false;
    }
}
```

Il faut cependant, pour que cette vérification soit possible, que l'attente programmée dans la méthode de retrait soit elle-même périodiquement interrompue. Nous utiliserons donc, à la place de la méthode wait(), la méthode :

```
public final native void wait(long timeout)
```

qui met en attente d'une notification pendant une période fixée par le paramètre (la durée est exprimée en millisecondes); cette méthode implémente donc une attente avec time-out.

Notre classe zone de transfert verra donc sa méthode retire() modifiée pour renvoyer -1 comme contenu en cas de disparition du producteur :

ZoneTransfert.java (2)

```
class ZoneTransfert
{
    private int contenu;
    private boolean occupé = false;
    private int cpt = 0;

    public synchronized int retire() throws InterruptedException
    {
        while (occupé == false && Producteur.vivant == true)
            wait(5000);
        if (Producteur.vivant == false) return -1;
        occupé = false;
        notify();
        return contenu;
    }

    public synchronized void dépose(int somme) throws InterruptedException { ... }
}
```

Il ne reste plus à chaque consommateur qu'à tester la somme obtenue pour savoir s'il doit encore poursuivre ou pas :

Consommateur.java (3)

```
class Consommateur extends Thread
{ ...
    public Consommateur (ZoneTransfert t, String n, int nc, Thread pr) { ... }

    public void run()
    {
        for (int i=0; i< nbreCheques; i++)
        {
            try
            {
                int sommeDéposée = transit.retire();
                if (sommeDéposée == -1)
                {
                    System.out.println("## " + nomC +

```

```
        " : le producteur est parti - je m'en vais aussi ...");  
    stop();  
}  
else  
    System.out.println("Le consommateur " + nomC + " retire  
                      " + sommeDeposée + " Euros" + " (chèque n°" + (i+1) + ")");  
}  
catch (InterruptedException e) {}  
}  
System.out.println("Le consommateur " + nomC + " est parvenu à terminer son travail");  
}  
}  
}
```

Une exécution donnera :

```
Je suis le producteur et le thread : Distributeur  
Les threads sont lancés !!!!  
>>>> depot n° 1  
Le producteur Distributeur dépose 14312 Euros  
Le consommateur Superman retire 14312 Euros (chèque n°1)  
>>>> depot n° 2  
Le producteur Distributeur dépose 21489 Euros  
Le consommateur Vilvens retire 21489 Euros (chèque n°1)  
>>>> depot n° 3  
Le producteur Distributeur dépose 35796 Euros  
Le consommateur Superman retire 35796 Euros (chèque n°2)  
>>>> depot n° 4  
Le producteur Distributeur dépose 13563 Euros  
Le consommateur Vilvens retire 13563 Euros (chèque n°2)  
Le consommateur Vilvens est parvenu à terminer son travail  
>>>> depot n° 5  
Le producteur Distributeur dépose 34567 Euros  
Le consommateur Superman retire 34567 Euros (chèque n°3)  
>>>> depot n° 6  
Le producteur Distributeur dépose 40498 Euros  
Le consommateur Bob Morane retire 40498 Euros (chèque n°1)  
>>>> depot n° 7  
Le producteur Distributeur dépose 19377 Euros  
Le consommateur Superman retire 19377 Euros (chèque n°4)  
Il n'y a plus de chèques !!!  
Threads encore en attente :  
Il y a 4 threads encore actifs  
Thread n° 0 = main  
Thread n° 1 = Distributeur  
Thread n° 2 = Superman  
Thread n° 3 = Bob Morane  
Le producteur a fini !!!!  
** Bob Morane : le producteur est parti - je m'en vais aussi ...  
** Superman : le producteur est parti - je m'en vais aussi ...
```

12. Le monitoring des threads sous Netbeans 5.5 ou 6.*

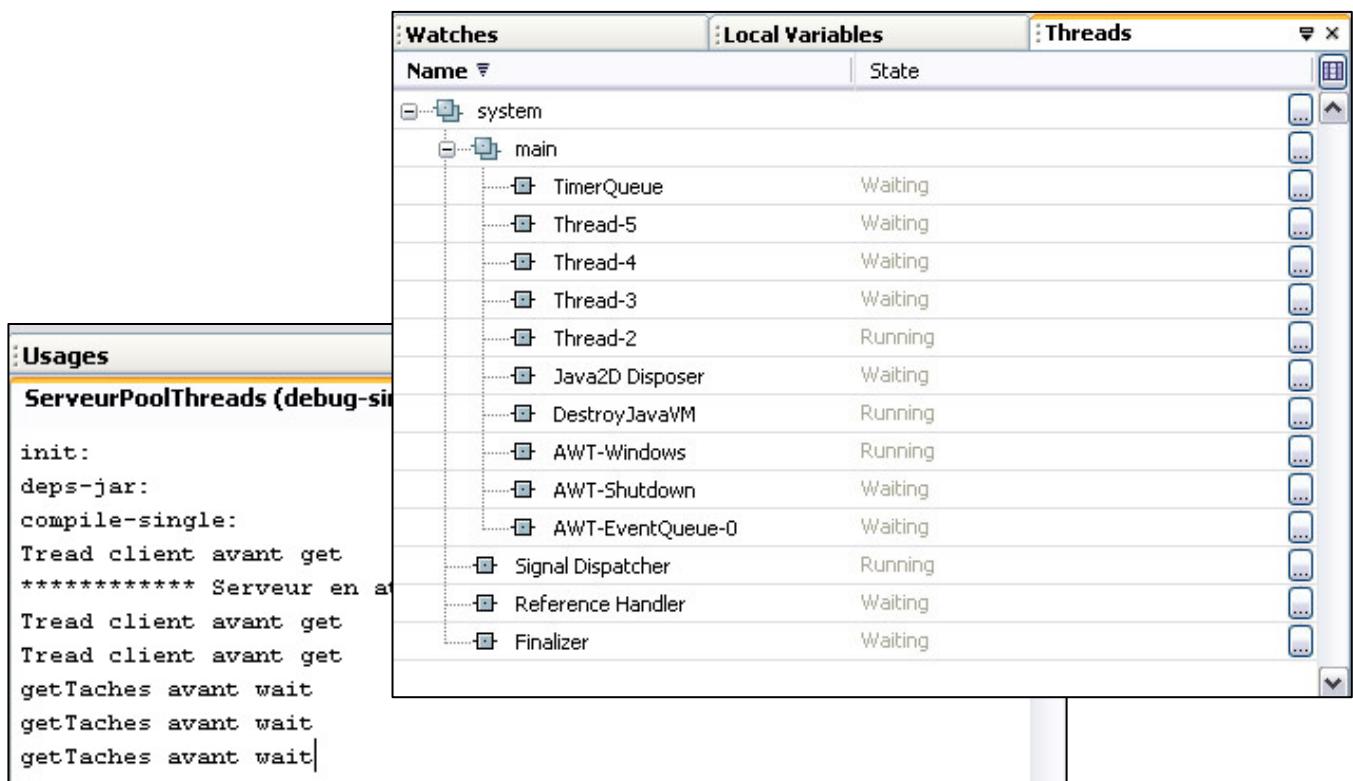
Dans le contexte de l'EDI Netbeans, il est possible de visualiser les threads intervenant dans une application à condition de travailler en mode Debug. Pour ce faire, il faut tout d'abord faire apparaître la fenêtre nécessaire par :

Window → Debugging → Threads

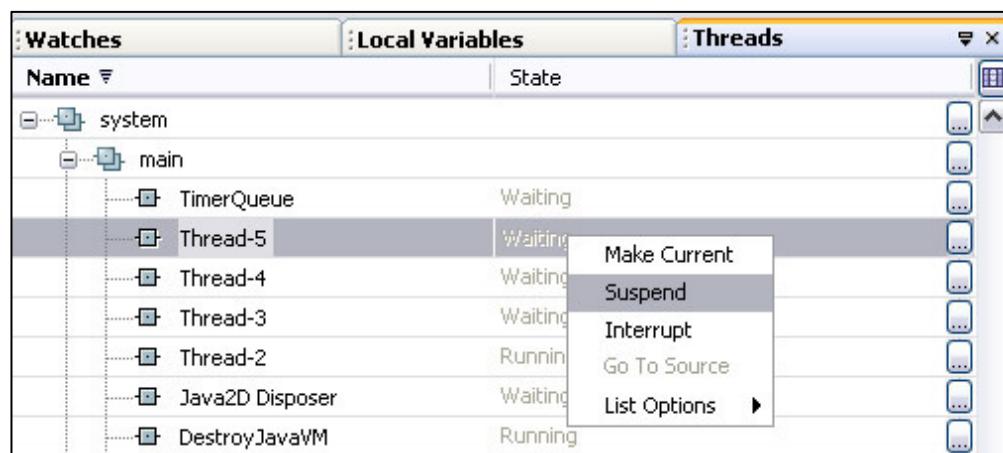
Ensuite, on peut lancer l'exécution de l'application en mode Debug par :

<clic droit> → Debug File

Pour une application exemple faisant tourner un serveur en modèle à pool de threads (voir chapitre consacré aux communications réseaux), on obtient alors



Il est possible aussi d'interagir sur les threads pour les arrêter, les suspendre, etc :



13. Les démons

Tous les threads dont nous avons parlé ici sont des ***user threads*** : l'application qui les a lancés ne se termine que lorsque tous ces threads sont eux mêmes terminés.

Il existe une autre catégorie : les ***deamon threads***. Ils correspondent bien à leurs homologues du même nom du monde UNIX : ce sont donc des threads en attente de demandes de services d'autres threads. Typiquement, le code comporte une boucle infinie comportant une attente de requête.

On transforme un thread en démon au moyen de la méthode :

```
public final void setDaemon(boolean on);
```

à qui l'on passe le paramètre true.

Bien logiquement, une application s'arrête lorsqu'elle ne comporte plus que des deamon threads, puisqu'il n'existe plus aucun thread qui pourrait demander un service ...

14. La communication entre les threads : les "Pipes Streams"

On conçoit aisément que les flux constituent un excellent organe de communication.. Nous allons donc ici construire un canal de communication entre deux threads, la sortie de l'un constituant l'entrée de l'autre.

14.1 Un thread et un flux de sortie

Nous allons tout d'abord créer une classe thread appelée *RandGenerator*

- ◆ chargé de générer 10 nombres aléatoires;
- ◆ dont le constructeur réclame un flux de sortie sur lequel le thread écrira :

```
public RandGenerator (OutputStream cible)
```

- ◆ dont la méthode run()
 - instanciera un flux de haut niveau sur le flux de base passé à son constructeur;
 - générera les nombres aléatoires en utilisant la méthode de classe **random()** de la classe Math de java.lang;
 - écrira ces nombres sur le flux de haut niveau.

Le thread est instancié et lancé dans le programme suivant, avec un flux fichier comme sortie :

PipeAndThread.java

```
import java.io.*;  
  
class RandGenerator extends Thread  
{  
    OutputStream sortie;  
  
    public RandGenerator (OutputStream cible)  
    {  
        sortie = cible;  
    }
```

```

public void run()
{
    double r;
    int n=0;

    System.out.println("Démarrage du thread de génération !");
    DataOutputStream s = new DataOutputStream(sortie);
    while (n<10)
    {
        r = Math.random();
        try { s.writeDouble(r); }
        catch (IOException e){}
        n++;
    }
}

// -----
public class PipeAndThread
{
    public static void main(String args[])
    {
        FileOutputStream fos;
        try
        {
            fos = new FileOutputStream("e:\\java-application\\PipeAndThread\\aleatoires.data");
            new RandGenerator(fos).start();
        }
        catch (IOException e) {}
    }
}

```

Le fichier aleatoires.data contient donc à présent 10 nombres réels aléatoires. Il ne reste donc plus qu'à lire ce fichier ...

14.2 Deux threads communicant par fichier

Créons à présent une classe thread cousine de la première, si ce n'est qu'elle réclame pour être instanciée un flux d'entrée. Nommons-la *RandExtractor*. On s'en doute, la sortie du premier thread constituera l'entrée du second : la communication entre ces deux threads est donc assurée via un fichier :

PipeAndThread.java (2)

```

import java.io.*;

class RandGenerator extends Thread
{
    ...
}

// -----

```

```

class RandExtractor extends Thread
{
    InputStream entree;
    public RandExtractor (InputStream source)
    {
        entree = source;
    }

    public void run()
    {
        int n=0;
        double rL;

        System.out.println("Démarrage du thread de lecture !");
        DataInputStream e = new DataInputStream(entree);

        while (n<10)
        {
            try
            {
                rL = e.readDouble();
                System.out.println("Lecture n° " + (n+1) + " => " + rL);
            }
            catch (IOException ex){}
            n++;
        }
    }
}

// -----
public class PipeAndThread
{
    public static void main(String args[])
    {
        FileOutputStream fos;
        FileInputStream fis;
        RandGenerator produc;
        try
        {
            fos = new FileOutputStream("e:\\java-application\\PipeAndThread\\aleatoires.data");
            fis = new FileInputStream("e:\\java-application\\PipeAndThread\\aleatoires.data");
            produc = new RandGenerator(fos);
            produc.setPriority(Thread.MAX_PRIORITY);
            produc.start();
            new RandExtractor(fis).start();
        }
        catch (IOException e) {}
    }
}

```

L'exécution donnera :

```
Démarrage du thread de génération !
Démarrage du thread de lecture !
Lecture n° 1 => 0.05271317390348451
Lecture n° 2 => 0.7216474311998509
Lecture n° 3 => 0.7728677788722674
Lecture n° 4 => 0.8532439962719236
Lecture n° 5 => 0.7135080132319932
Lecture n° 6 => 0.042862042865664085
Lecture n° 7 => 0.36922139381248087
Lecture n° 8 => 0.08392178089090097
Lecture n° 9 => 0.8068384636955137
Lecture n° 10 => 0.11816550859364483
```

On remarquera que le thread producteur a été lancé avec la priorité maximale afin que le thread consommateur trouve effectivement quelque chose à lire. Cette pratique n'est certes pas exemplaire, mais elle suffit à illustrer la communication – on se référera au paragraphe consacré aux moniteurs de threads pour de meilleurs moyens de synchronisation.

L'essentiel est ailleurs : le fichier ainsi créé est en fait à vocation temporaire et il serait d'ailleurs logique de l'effacer en fin de communication. Mais il y a plus simple ...

14.3 Deux threads communicant par tube (pipe)

Les tubes (*pipes* en anglais) sont des canaux de communications entre threads. Ils ne sont pas spécifiques à la programmation Java, puisque la programmation sous Windows XP/2000/NT/95/98 les connaît également sous le nom de "tubes nommés" pour la programmation réseau. Ils constituent une alternative intéressante à l'utilisation d'un fichier temporaire de communication, illustrée ci-dessus, spécialement dans le cas de communications réseaux.

En Java, les *pipes* sont matérialisés par deux classes, **PipedInputStream** et **PipedOutputStream**, dérivées respectivement d'InputStream et d'OutputStream, selon que le canal fait office d'entrée ou de sortie. Leurs caractéristiques sont évidemment "symétriques". Ainsi en est-il des constructeurs :

```
public PipedInputStream()
public PipedInputStream(PipedOutputStream src) throws IOException
```

et

```
public PipedOutputStream()
public PipedOutputStream(PipedInputStream snk) throws IOException
```

Le point important est que ces flux *pipes* seront toujours instanciés **par paire** : l'un se connecte en fait sur l'autre, l'ordre n'ayant pas d'importance.

Ils seront toujours associés à **deux threads**, l'un d'écriture sur le canal et l'autre de lecture sur ce même canal.

Ils possèdent respectivement les méthodes bien connues :

```
public synchronized int read(byte b[], int off, int len) throws IOException
public void write(byte b[], int off, int len) throws IOException
```

ainsi que leurs consœurs – mais, en pratique, ce sera plutôt la tâche d'un flux de haut niveau de réaliser des opérations de lecture ou d'écriture convenable.

Le programme suivant ressemble au précédent, mais en utilisant la technique des *pipes* :

PipeAndThread.java (3)

```
import java.io.*;

class RandGenerator extends Thread
{
    OutputStream sortie;
    public RandGenerator (OutputStream cible) {sortie = cible; }

    public void run() { ... }
}

// -----
class RandExtractor extends Thread
{
    InputStream entree;
    public RandExtractor (InputStream source) {entree = source; }

    public void run() { .... }
}

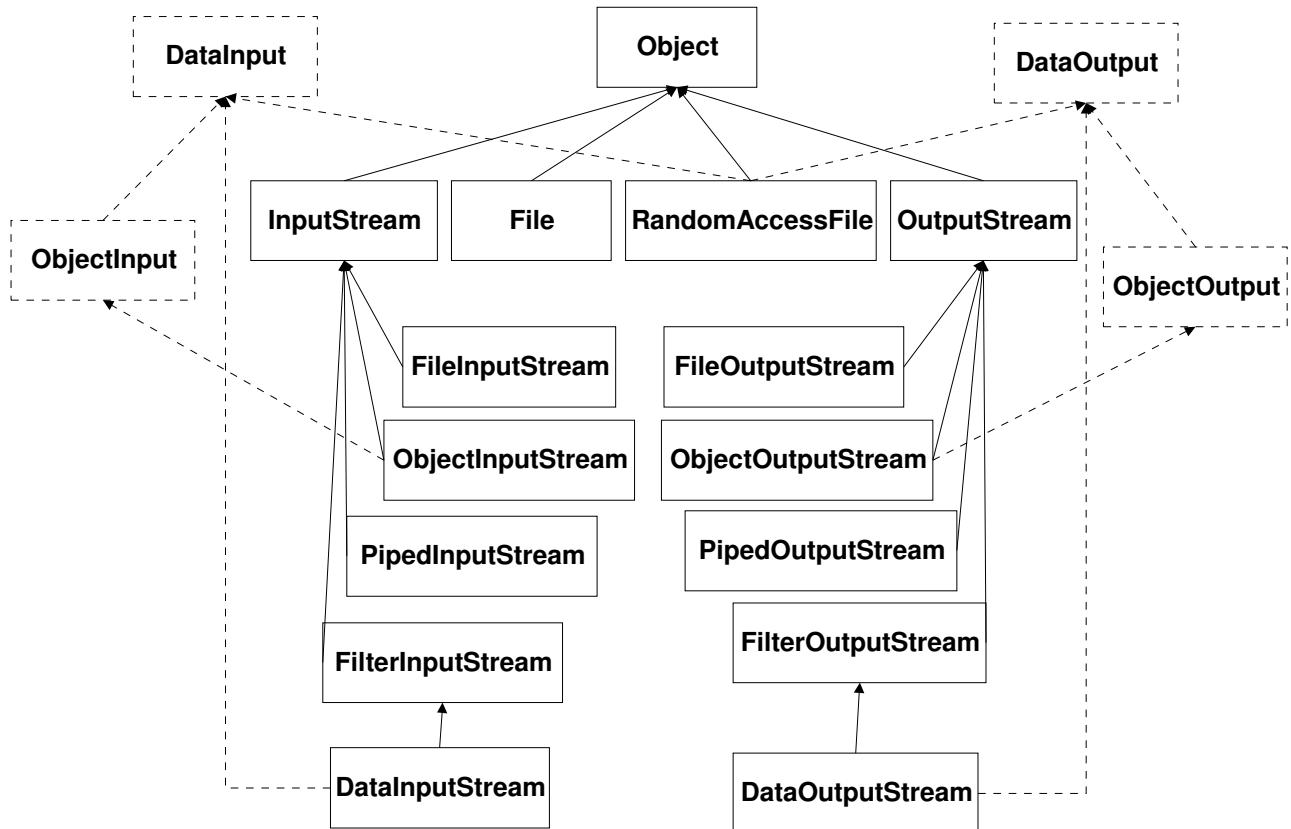
// -----
public class PipeAndThread
{
    public static void main(String args[])
    {
        PipedOutputStream pos;
        PipedInputStream pis;
        RandGenerator produc;
        try
        {
            pos = new PipedOutputStream();
            pis = new PipedInputStream(pos);
            produc = new RandGenerator(pos);
            produc.setPriority(Thread.MAX_PRIORITY);
            produc.start();
            new RandExtractor(pis).start();
        }
        catch (IOException e) {}
    }
}
```

La sortie à l'exécution est identique à celle du programme précédent.

Remarques

1) On peut sérialiser sur un pipe, donc utiliser des ObjectOutputStream et ObjectInputStream sur des PipedOutputStream et PipedInputStream.

2) Nous pouvons donc compléter la hiérarchie des flux orientés bytes, hiérarchie évoquée au chapitre VII :



15. Les threads prédéfinis du framework AWT/Swing

Les composants graphiques que nous avons l'habitude de manipuler (fenêtres, boîtes de dialogues, composants divers) constituent en fait une couche abstraite s'appuyant sur une couche d'implémentation comportant notamment les threads : on parle encore de DRL (**Dynamic Run-time Layer**). Autrement dit, la plate-forme Java a prévu un système multithread pour gérer de manière efficace les GUIs sans que le développeur ait à s'occuper de cet aspect. On peut d'ailleurs se rendre compte de l'existence de ces threads en exécutant une simple application GUI et en observant le monitoring des threads correspondant :

Name	State
system	
main	
Java2D Disposer	Waiting
AWT-Shutdown	Waiting
AWT-Windows	Running
AWT-EventQueue-0	Waiting
TimerQueue	Waiting
DestroyJavaVM	Running
Reference Handler	Waiting
Finalizer	Waiting
Signal Dispatcher	Running

On constate donc l'existence

- ◆ d'un groupe main de threads avec un thread principal qui exécute la méthode main;
- ◆ de deux threads de gestion de la mémoire selon le mécanisme du Garbage collector ;
- ◆ d'un thread chargé de gérer le traitement des signaux asynchrones provenant du système d'exploitation hôte.

Ce qui se passe réellement en termes d'implémentations de threads dépend en partie de l'environnement (notamment en ce qui concerne la création de threads supplémentaires pour certaines actions, threads démons ou non). Cependant, les principes sont les mêmes sur toutes les machines hôtes.

En fait, dans le cas d'une application utilisant un GUI, il existe plutôt un groupe de threads avec parmi eux le thread main (une simple instance de la classe Thread) et surtout un thread nommé AWT-EventQueue-0, instance de la classe **EventDispatchThread** (classe privée du package java.awt et qui hérite de Thread). Le rôle de ce thread est de gérer l'interface graphique et ceci de manière exclusive : **tout ce qui concerne le GUI doit être exécuté dans ce thread et de manière séquentielle** – de cette manière, il n'est pas nécessaire de s'embarrasser de méthodes synchronized, puisque les actions sur le GUI ne sont pas réalisées en parallèle. Comment cette sequentialité est-elle assurée ?

En fait, tout événement graphique (un appui sur un bouton, une prise ou une perte de focus, etc) est tout d'abord matérialisé par un "événement natif" du système d'exploitation hôte (en pratique, des "messages Windows", des XEvents sous Unix, etc ...); une composante de l'AWT, clairement dépendante du système réel sur lequel l'application fonctionne, prend cet événement natif en compte et interface une autre composante de l'AWT, cette fois indépendante du système réel, et dont le rôle est de créer un objet de type AWTEvent, qui est placé dans une file instance de **EventQueue** (package java.awt) au moyen de la méthode

```
public void postEvent(AWTEvent theEvent)
```

de celle-ci. Le thread de dispatch n'a donc plus qu'à prendre dans la file et à appeler la méthode du composant concerné

```
protected void processEvent(AWTEvent e)
```

qui elle-même aiguille sur une méthode **processXXXEvent()** (comme par exemple processFocusEvent (FocusEvent e)) ce qui a pour effet d'aviser tous les **listeners** intéressés par ces événements pour le composant en question.

En fait, c'est ainsi que les choses se passent à l'origine, dans le JDK 1.1. Actuellement, la méthode postEvent() est obsolète et on utilise l'EventQueue en utilisant la méthode

```
public static void invokeLater(Runnable runnable)
```

qui demande au EventDispatchThread d'exécuter la méthode run() de l'objet Runnable lorsque la file aura été vidée des événements qui s'y trouvaient déjà, évitant ainsi tout risque de perturbation. C'est donc de cette manière qu'un thread quelconque peut faire exécuter une action sur le GUI par le thread dispatcher. On peut encore savoir qu'il existe

```
public static boolean isDispatchThread()
```

- cette méthode statique de EventQueue permet de savoir si le thread courant est le thread dispatcher, auquel cas on peut exécuter immédiatement le traitement au lieu de le placer dans la file.

16. Les timers

16.1 Deux classes timers

Il est assez courant de souhaiter invoquer un thread de manière régulière, en fonction de l'écoulement d'une période de temps. Les objets qui permettent cela sont des objets instances de la classe **Timer**. Mais, en fait, il en existe plusieurs dans les librairies de Java : nous allons nous contenter de deux d'entre elles :

- ◆ dans **java.util** : public class **Timer** extends Object
- ◆ dans **javax.swing** : public class **Timer** extends Object implements Serializable

La première classe est plus générale et laisse le développeur tout gérer. La seconde est plus du style "factory" puisqu'elle est conçue d'une manière telle que

- ◆ tous les timers instances de cette classe partagent le même thread (celui-ci se "schedule" entre les diverses tâches);
- ◆ les événements de type ActionEvent qu'elle génère sont en fait placés dans l'EventQueue, et donc traités par l'EventDispatchThread – il n'est donc pas nécessaire d'utiliser la méthode invokeLater().

16.2 Le timer de java.util

Nous allons tout d'abord utiliser le premier type de timer dans une application qui ne nécessite pas de précaution particulière en termes d'EventQueue, c'est-à-dire une application console. En nous inspirant d'un exemple rencontré dans un certain cours de threads en C sous Unix ;-), imaginons que nous simulions un supermarché rudimentaire dans lequel

- ◆ les clients (en nombre fixé) sont simulés par des threads;
- ◆ le gérant est le thread principal;
- ◆ le manutentionnaire est un thread (ou du moins exécuté par un thread ;-) qui s'active périodiquement pour voir si il n'y a pas de rupture de stock ou autre chose qui réclame son intervention.

Le résultat serait donc du type suivant :

Thread-0> Je vais acheter 5 articles
Thread-1> Je vais acheter 4 articles
Thread-2> Je vais acheter 3 articles
Thread-3> Je vais acheter 2 articles
Thread-4> Je vais acheter 1 articles
Thread-4> J'achète un article (55)
Thread-1> J'achète un article (215)
* Manutentionnaire> quelque chose à faire ?
Thread-4> J'ai acheté 1 articles
Thread-1> J'achète un article (250)
* Manutentionnaire> quelque chose à faire ?
Thread-2> J'achète un article (230)
Thread-1> J'achète un article (50)
* Manutentionnaire> quelque chose à faire ?
...
Thread-0> J'achète un article (115)
Thread-2> J'ai acheté 3 articles
* Manutentionnaire> quelque chose à faire ?
* Manutentionnaire> quelque chose à faire ?
Thread-3> J'achète un article (255)
* Manutentionnaire> quelque chose à faire ?
...
* Manutentionnaire> quelque chose à faire ?
...
Thread-0> J'achète un article (95)
* Manutentionnaire> quelque chose à faire ?
Thread-0> J'ai acheté 5 articles
Tous les clients ont terminé
On ferme ...

L'exécution en mode debug montre le monitoring des threads :

Name	State
system	
main	
main	Waiting
Thread-1	Sleeping
Thread-3	Sleeping
Thread-0	Sleeping
Thread-2	Sleeping
Thread-4	Sleeping
Timer-0	Waiting
Reference Handler	Waiting
Finalizer	Waiting
Signal Dispatcher	Running

Les threads client sont élémentaires :

Client.java

```

package supermarche;

public class Client extends Thread
{
    private int borneInferieure, borneSuperieure, multipleDeclenchement, tempsPause;
    private int nombreArticlesAAcheter;
    private int nombreProduit;

    public Client(int bi, int bs, int md, int tp, int nba)
    {
        borneInferieure=bi; borneSuperieure=bs;
        multipleDeclenchement = md; tempsPause = tp;
        nombreArticlesAAcheter = nba; nombreProduit = -1;
        System.out.println(this.getName() + "> Je vais acheter " + nombreArticlesAAcheter
                           + " articles");
    }

    @Override
    public void run()
    {
        for (int i=0; i<nombreArticlesAAcheter; )
        {
            Double dr = new Double(borneInferieure +
                Math.random()*(borneSuperieure - borneInferieure));
            nombreProduit = dr.intValue();
            if (nombreProduit % multipleDeclenchement == 0)
            {
                System.out.println(this.getName() + "> J'achète un article (" +
                                   nombreProduit + ")");
                i++;
            }
            try
            {
                Thread.sleep(tempsPause*1000);
            }
            catch (InterruptedException e)
            {
                System.out.println("Erreur de thread interrompu : " + e.getMessage());
            }
        }
        System.out.println(this.getName() + "> J'ai acheté "+nombreArticlesAAcheter
                           + " articles");
    }
}

```

Passons au manutentionnaire. Dans sa version la plus simple, un objet **Timer** s'instancie avec son constructeur par défaut (mais il existe des versions polymorphes avec un flag pour être démon et un nom à donner au thread).

Ce qui fait l'objet du timer est un objet **TimerTask** (du package `java.util`) qui n'est pas exactement un thread mais plutôt une implémentation de `Runnable` (et donc susceptible d'être exécuté par un thread créé par l'objet `Timer`). Dans notre cas, il suffira donc de définir notre manutentionnaire comme étant une classe héritée de ce `TimerTask` :

Manutentionnaire.java

```
package supermarche;

import java.util.*;

public class Manutentionnaire extends TimerTask
{
    @Override
    public void run()
    {
        System.out.println( "* Manutentionnaire> quelque chose à faire ?" );
    }
}
```

Le timer peut s'utiliser de deux manière, décrite par sa méthode **schedule()** qui établit la relation avec le `TimerTask` à exécuter :

- ◆ ou bien l'on souhaite programmer l'exécution d'une tâche après un certain temps ou à une certaine date :

```
public void schedule(TimerTask task, long delay)
public void schedule(TimerTask task, Date time)
```

- ◆ ou bien on souhaite voir une tâche être effectuée de manière répétitive, après un certain temps d'initialisation précisé comme troisième paramètre :

```
public void schedule(TimerTask task, long delay, long period)
public void schedule(TimerTask task, Date firstTime, long period)
```

Le magasin (donc le gérant) n'aura donc qu'à

- ◆ créer ses clients et les faire démarrer;
- ◆ enclencher un timer en lui donnant le manutentionnaire à séquencer;
- ◆ attendre la fin des clients pour fermer le magasin, ce qu'il fera au moyen d'une méthode de synchronisation

```
public final void join() throws InterruptedException
public final void join(long millis) throws InterruptedException
public final void join(long millis, int nanos) throws InterruptedException
```

- ◆ arrêter le thread manutentionnaire en annulant le timer au moyen de la méthode

```
public void cancel()
```

Donc :

Magasin.java

```
package supermarche;

public class Magasin
{
    private static final int NOMBRE_CLIENTS = 5;
    public static void main(String[] args)
    {
        Client[] c = new Client[NOMBRE_CLIENTS];
        for (int nt=0; nt<NOMBRE_CLIENTS; nt++)
        {
            c[nt] = new Client(10, 300, 5, 5, NOMBRE_CLIENTS - nt);
            c[nt].start();
        }
        Timer t = new Timer();
        Manutentionnaire m = new Manutentionnaire();
        t.schedule(m, 2000, 5000 );
        for (int nt=0; nt<NOMBRE_CLIENTS; nt++)
        {
            try
            {
                c[nt].join(0);
            }
            catch (InterruptedException e)
            {
                System.out.println("Erreur de thread interrompu : " + e.getMessage());
            }
        }
        System.out.println("Tous les clients ont terminé");
        t.cancel();
        System.out.println("On ferme ...");
    }
}
```

Le résultat est bien conforme à l'attente. Passons à la version swing ...

16.3 Le timer de javax.swing

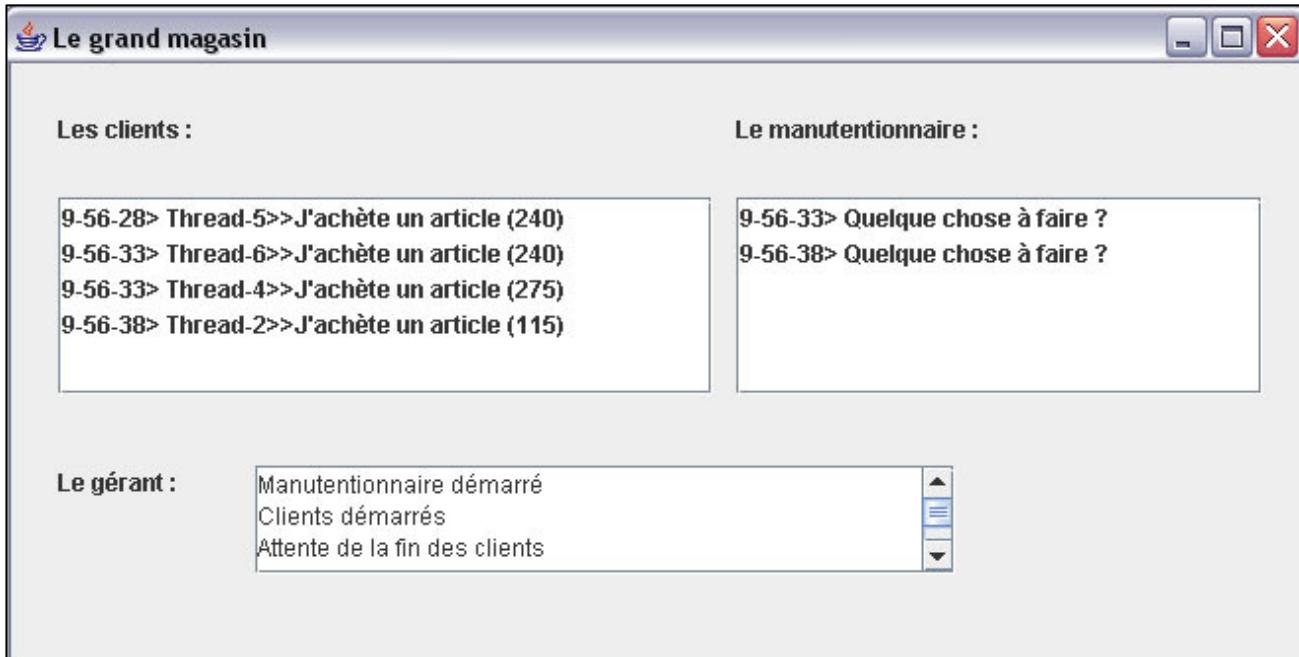
La classe Timer de javax.swing est certainement d'un niveau d'abstraction supérieur, puisque qu'elle considère qu'après chaque période de temps précisé dans son constructeur, elle doit délivrer une notification à l'ActionListener également désigné dans son constructeur. :

```
public Timer (int delay, ActionListener listener)
```

Ce qui fait l'objet de la tâche à répéter périodiquement doit donc être codé dans la méthode actionPerformed() de l'ActionListener considéré. Simple ...

16.4 Une application GUI threadée : principe

Reprenons le thème du supermarché pour envisager l'application GUI simple suivante :



Un minimum de réflexion, et quelques velléités de généricité, nous conduisant à la conception suivante :

- ◆ les clients (en nombre fixé) sont simulés par des threads (classe **Client**);
- ◆ le gérant est matérialisé par le fenêtre principale (classe **FenApp**); c'est lui qui est lancé au démarrage de l'application par un invokeLater() de l'EventQueue – c'est donc bien l'**EventDispatchThread** qui exécute ses méthodes;
- ◆ le manutentionnaire (classe **Manutentionnaire**) est l'actionListener qui s'active périodiquement pour voir si il n'y a pas de rupture de stock ou autre chose qui réclame son intervention;
- ◆ les clients, le manutentionnaire et même le gérant, seront créés en leur passant la référence d'une zone d'affichage (une JList pour les deux premiers, une JTextArea pour le dernier) – disons que l'interface **Afficheur** correspond à une telle zone, avec pour seule méthode addTexte(String x); deux implémentations en seront les classes **AfficheurListe** et **AfficheurTextArea**;
- ◆ pour faire réaliser l'affichage dans le GUI par le thread EventDispatchThread, un intervenant de l'application créera un Runnable (disons qu'il s'agit d'une instance *ta* de classe **TacheAfficheur**) dont la méthode run() appelle la méthode addTexte() de l'afficheur dont il possède la référence, puis placera cette tâche dans l'EventQueue au moyen des appels enchaînés :

```
Toolkit.getDefaultToolkit().getSystemEventQueue().invokeLater(ta);
```

Explications : la classe **Toolkit** (du package java.awt) est une classe abstraite mère des classes chargées de réaliser le lien entre la couche abstraite des GUIs Java et la plate-forme réelle où ces GUIs sont réalisés. Elle possède une méthode

```
public static Toolkit getDefaultToolkit()
```

dont le rôle est de fournir l'objet courant (elle peut être fournie par la propriété système "awt.toolkit" ou, si celle-ci n'est pas définie, elle sera une instance de "sun.awt.motif.MToolkit"). Avec un tel rôle, on ne s'étonnera pas de trouver aussi le moyen d'accéder à l'**EventQueue** au moyen de :

```
public final EventQueue getSystemEventQueue()
```

dont il ne restera plus qu'à appeler la méthode invokeLater().

16.5 Une application GUI threadée : implémentation

L'implémentation des principes ci-dessus ne pose aucun problème majeur.

1) Tout d'abord, nous définissons les afficheurs :

Afficheur.java

```
package supermarchegui;

/**
 * @author Vilvens
 */

public interface Afficheur
{
    void addTexte (String x);
}
```

AfficheurListe.java

```
package supermarchegui;

import javax.swing.DefaultListModel;
import javax.swing.JList;

/**
 * @author Vilvens
 */

public class AfficheurListe implements Afficheur
{
    private JList liste;
    public AfficheurListe (JList l)
    {
        liste = l;
    }
    public void addTexte(String x)
    {
        DefaultListModel dlm = (DefaultListModel)this.liste.getModel();
        dlm.addElement(x);
        //throw new UnsupportedOperationException("Not supported yet.");
    }
}
```

AfficheurTextArea.java

```
package supermarchegui;

import javax.swing.JTextArea;

/**
 * @author Vilvens
 */
public class AfficheurTextArea implements Afficheur
{
    private JTextArea aireTexte;
    public AfficheurTextArea (JTextArea ta)
    {
        aireTexte = ta;
    }

    public void addTexte(String x)
    {
        aireTexte.append(x + System.getProperty("line.separator"));
        //throw new UnsupportedOperationException("Not supported yet.");
    }
}
```

2) La classe Runnable représentant une tâche à réaliser sur un afficheur réclame simplement comme second paramètre le texte à afficher :

TacheAfficheur.java

```
package supermarchegui;

/**
 *
 * @author Vilvens
 */
public class TacheAfficheur implements Runnable
{
    private Afficheur zoneAffichageTache;
    private String texteTache;
    public TacheAfficheur (Afficheur a, String t)
    {
        zoneAffichageTache = a;
        texteTache = t;
    }
    public void run()
    {
        zoneAffichageTache.addTexte(texteTache);
        //throw new UnsupportedOperationException("Not supported yet.");
    }
}
```

3) Les threads clients utilisent les services d'un afficheur et créent une tâche pour l'EventQueue :

Client.java

```
package supermarchegui;

import java.awt.Toolkit;
import java.util.Calendar;

/**
 * @author Vilvens
 */

public class Client extends Thread
{
    private int borneInferieure, borneSuperieure, multipleDeclenchement, tempsPause;
    private int nombreArticlesAAcheter, nombreProduit;
    private Afficheur zoneAffichage;

    public Client(int bi, int bs, int md, int tp, int nba, Afficheur a)
    {
        borneInferieure=bi; borneSuperieure=bs;
        multipleDeclenchement = md; tempsPause = tp;
        nombreArticlesAAcheter = nba; nombreProduit = -1;
        zoneAffichage = a;
        System.out.println(Thread.currentThread().getName() + "> Je vais acheter " +
            nombreArticlesAAcheter + " articles");
    }

    @Override
    public void run()
    {
        for (int i=0; i<nombreArticlesAAcheter; )
        {
            Double dr = new Double(borneInferieure +
                Math.random()*(borneSuperieure - borneInferieure));
            nombreProduit = dr.intValue();

            if (nombreProduit % multipleDeclenchement == 0)
            {
                System.out.println(this.getName() + "> J'achète un article (" +
                    nombreProduit + ")");
                i++;
            }

            Calendar maintenant = Calendar.getInstance();
            int h = maintenant.get(Calendar.HOUR_OF_DAY);
            int m = maintenant.get(Calendar.MINUTE);
            int s = maintenant.get(Calendar.SECOND);
        }
    }
}
```

```

TacheAfficheur ta = new TacheAfficheur(
    zoneAffichage,
    new String (h + "-" + m + "-" +s)+"> "
        + this.getName()+">>J'achète un article (" +
        nombreProduit + ")"
);
Toolkit.getDefaultToolkit().getSystemEventQueue().
    invokeLater(ta);
}
try
{
    Thread.sleep(tempesPause*1000);
}
catch (InterruptedException e)
{
    System.out.println("Erreur de thread interrompu : " + e.getMessage());
}
}
System.out.println(this.getName()+">> J'ai acheté "+nombreArticlesAAcheter
    +" articles");
}
}
}

```

4) La fenêtre principale initialise ses divers afficheurs et lance les threads clients. Et, bien sûr, lance un timer avec un manutentionnaire comme ActionListener :

FenApp.java

```

package supermarchegui;

import java.awt.Toolkit;
import javax.swing.DefaultListModel;

/**
 * @author Vilvens
 */
public class FenApp extends javax.swing.JFrame
{
    private static final int NOMBRE_CLIENTS = 5;

    private AfficheurListe alm, alc;
    private AfficheurTextArea ata;

    public FenApp()
    {
        System.out.println("FenApp : Thread actif = " + Thread.currentThread().getName());
        System.out.println(" instance de = " + Thread.currentThread().getClass().getName());
        initComponents();
        jList2.setModel(new DefaultListModel() );
        alm = new AfficheurListe(this.jList2);
    }
}

```

```

javax.swing.Timer t = new javax.swing.Timer(5000, new Manutentionnaire(alm));
t.start();
// Ce qu'il faut éviter : this.jTextArea1.append("Manutentionnaire démarré\n");
ata = new AfficheurTexteArea (jTextArea1);
TacheAfficheur ta = new TacheAfficheur( ata, "Manutentionnaire démarré");
Toolkit.getDefaultToolkit().getSystemEventQueue().invokeLater(ta);

jList1.setModel(new DefaultListModel() );
alc= new AfficheurListe(this.jList1);
Client[] c = new Client[NOMBRE_CLIENTS];
for (int nt=0; nt<NOMBRE_CLIENTS; nt++)
{
    c[nt] = new Client(10, 300, 5, 5, NOMBRE_CLIENTS - nt, alc);
    c[nt].start();
}
ta = new TacheAfficheur(ata, "Clients démarrés");
Toolkit.getDefaultToolkit().getSystemEventQueue().invokeLater(ta);
ta = new TacheAfficheur(ata, "Attente de la fin des clients");
Toolkit.getDefaultToolkit().getSystemEventQueue().invokeLater(ta);
}

private void initComponents() {...}

public static void main(String args[])
{
    System.out.println("Thread actif = " + Thread.currentThread().getName());
    System.out.println(" instance de = " +
        Thread.currentThread().getClass().getName());
    java.awt.EventQueue.invokeLater(new Runnable()
    {
        public void run() { new FenApp().setVisible(true); }
    });
}

private javax.swing.JList jList1;
private javax.swing.JList jList2;
private javax.swing.JScrollPane jScrollPane1;
private javax.swing.JScrollPane jScrollPane2;
private javax.swing.JScrollPane jScrollPane3;
private javax.swing.JTextArea jTextArea1;
...
}

```

5) Et finalement le manutentionnaire lui-même se révèle fort simple :

Manutentionnaire.java

```
package supermarchegui;
```

```
import java.awt.event.ActionEvent;
```

```

import java.awt.event.ActionListener;
import java.util.Calendar;
import java.util.Date;

/**
 * @author Vilvens
 */
public class Manutentionnaire implements ActionListener
{
    private Afficheur zoneAffichage;

    public Manutentionnaire (Afficheur a)
    {
        zoneAffichage = a;
    }
    public void actionPerformed(ActionEvent e)
    {
        //throw new UnsupportedOperationException("Not supported yet.");
        System.out.println( "* Manutentionnaire> quelque chose à faire ?" );
        Calendar maintenant = Calendar.getInstance();
        int h = maintenant.get(Calendar.HOUR_OF_DAY);
        int m = maintenant.get(Calendar.MINUTE);
        int s = maintenant.get(Calendar.SECOND);
        zoneAffichage.addTexte(new String (h + "-" + m + "-" +s)
            +"> Quelque chose à faire ?");
    }
}

```

17. Les applets et les threads

17.1 Une horloge qui ne s'affiche pas

Considérons l'exemple classique (et fourni par Sun) de l'applet qui réalise une horloge à affichage digital. On *pourrait* penser qu'il suffit de programmer une boucle d'affichage de l'heure dans la fonction de démarrage de l'applet. Cela donnerait quelque chose du genre :

pseudoHorloge.java

```

import java.applet.Applet;

import java.awt.Graphics;
import java.awt.Font;
import java.util.Date;

public class pseudoHorloge extends Applet
{
    private Font f = new Font("TimesRoman",Font.BOLD,24);
    private Date maintenant ;

    public void start()

```

```

{
    while (true)
    {
        maintenant = new Date();
        repaint();
        System.out.println("Et pourtant, je travaille ... : " + maintenant );
        try
        {
            Thread.sleep(1000);
        }
        catch (InterruptedException e) { }
    }

    public void paint(Graphics g)
    {
        g.setFont(f);
        g.drawString(maintenant .toString(),10,50);
    }
}

```

Quelques remarques :

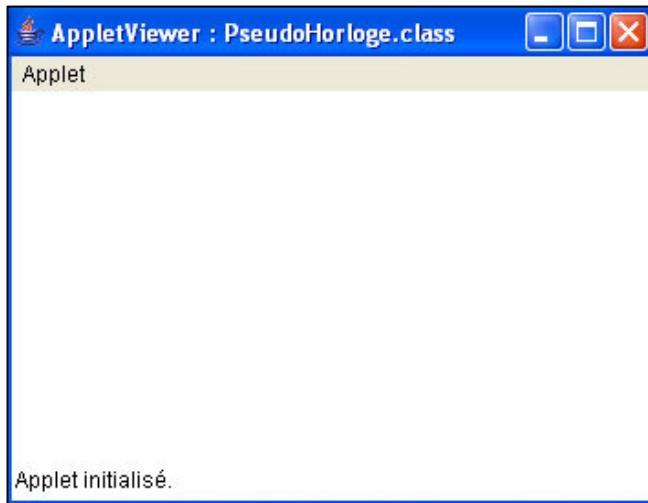
- ◆ on utilise la classe **Date**, définie dans le package `java.util`; pour rappel (chapitre VI), le constructeur sans argument prend la date système (plus exactement, la date de son allocation);
- ◆ l'applet demande de reconstruire son interface graphique (méthode `repaint()`) toutes les secondes – le thread principal reste en effet suspendu durant 1000 millisecondes;
- ◆ la méthode `sleep()` ne peut être utilisée sans le traitement de l'exception `InterruptedException`;
- ◆ accessoirement, on peut encore remarquer l'utilisation d'une police de caractère particulière : on utilise pour cela une instance de la classe `Font`, dont le constructeur

`public Font(String name, int style, int size)`

dont

- ♣ le premier paramètre est le nom de la police (par exemple : "TimesRoman");
- ♣ le deuxième est l'une des variables de classes `BOLD`, `ITALIC` ou `PLAIN`;
- ♣ le troisième est la taille.

Le seul problème est que cette applet n'affiche rien !



Pourtant, on peut se convaincre qu'elle travaille en regardant la console Java :

```
Et pourtant, je travaille ... : Wed Apr 27 18:02:55 CEST 2005  
Et pourtant, je travaille ... : Wed Apr 27 18:02:56 CEST 2005  
Et pourtant, je travaille ... : Wed Apr 27 18:02:57 CEST 2005  
Et pourtant, je travaille ... : Wed Apr 27 18:02:56 CEST 2005  
Et pourtant, je travaille ... : Wed Apr 27 18:02:57 CEST 2005  
Et pourtant, je travaille ... : Wed Apr 27 18:02:58 CEST 2005  
Et pourtant, je travaille ... : Wed Apr 27 18:02:59 CEST 2005  
...
```

Que se passe-t-il ? Tout simplement que *la méthode start() monopolise quasiment le processeur*, ne laissant pratiquement rien à la méthode paint(). Autrement dit, l'affichage n'a pas le temps d'être réalisé. On peut s'en convaincre en analysant les performances de notre applet au moyen d'un outil approprié quelconque montrant le temps d'exécution de chaque méthode : la méthode paint() n'a pratiquement droit à rien ☺ !

La solution à ce problème est simple si l'on pense au fonctionnement des threads : il faut que l'applet lance un thread complémentaire au thread principal. Ce dernier se chargera de l'affichage pendant que l'autre se réveillera à chaque seconde ...

17.2 Un thread dans une applet

Dans le cas où l'on a l'intention d'exécuter un thread au sein d'une applet (ce qui est nécessaire pour réaliser ces animations qui ont justifié le succès de Java auprès des concepteurs de pages WEB), le plus simple est de lui adjoindre *une variable membre qui est le thread* responsable de l'affichage de l'heure :

Thread thrHorloge;

La fonction à exécuter par ce thread sera, si l'on veut rester cohérent, une méthode de l'applet. Comment "passer" cette méthode au thread ? En fait, *l'applet va implémenter l'interface Runnable*, qui, pour rappel, ne comporte que la méthode

```
void run();
```

Cette méthode **run()** de Runnable est donc implémentée par l'applet : essentiellement, elle demande à l'applet de reconstruire son interface graphique (**repaint()**) toutes les secondes – le

thread reste en effet suspendu durant 1000 millisecondes. *L'applet servira donc à initialiser la variable membre target du thread.* En effet, celui-ci est instancié en lui passant l'applet comme paramètre. Plus précisément, la méthode **start()** de l'applet crée le thread et le lance. Le constructeur utilisé ici pour le thread réclame comme premier argument l'objet qui implémente l'interface Runnable, soit ici l'objet applet lui-même désigné par this :

```
thrHorloge = new Thread (this, "Horloge");
thrHorloge.start();
```

Le thread exécutera donc finalement, au sein de sa méthode run(), la méthode run() de l'applet ...

Tout ceci se passe tant que le thread n'est pas interrompu par la méthode stop() de l'applet. Cependant, comme dans le paragraphe précédent, il n'est plus question d'interrompre le thread au moyen de sa méthode **deprecated** stop() : comment alors indiquer au thread Horloge qu'il doit s'interrompre ? Le plus simple pour l'applet est

- ◆ d'annuler, au sein de sa propre méthode **stop()**, la référence du thread Horloge :

```
thrHorloge = null;
```

- ◆ d'obliger ce thread Horloge à constamment vérifier si il est bien le thread courant, ce qui se teste en utilisant la méthode de classe **currentThread()** qui renvoie précisément une référence sur ce thread courant – dans la négative, le thread Horloge terminera sa méthode run() normalement et sans heurt :

```
while (Thread.currentThread() == thrHorloge)
{
    repaint();
    try
    {
        thrHorloge.sleep(1000);
    }
    catch (InterruptedException e)
    {}
}
```

Pour que l'effet soit correct, il conviendra de déclarer la variable member thread comme étant volatile. L'applet s'écrit donc :

Heure.java

```
import java.applet.*;
import java.awt.*;
import java.util.*;

public class Heure extends Applet implements Runnable
{
    private volatile Thread thrHorloge;
```

```

public void start()
{
    if ( thrHorloge == null)
    {
        thrHorloge = new Thread (this, "Horloge");
        thrHorloge.start();
    }
}

public void stop()
{
    thrHorloge = null;
}

public void run()
{
    while (Thread.currentThread() == thrHorloge) // le thread est-il toujours là ?
    {
        repaint();
        try { thrHorloge.sleep(1000); }
        catch (InterruptedException e) {}
    }
}

public void paint(Graphics g)
{
    Font f = new Font("TimesRoman", Font.BOLD, 30);
    g.setFont(f);
    g.setColor(Color.red);
    Calendar maintenant = Calendar.getInstance();
    g.drawString( String.valueOf(maintenant.get(Calendar.HOUR_OF_DAY)) + ":" +
                  String.valueOf(maintenant.get(Calendar.MINUTE)) + ":" +
                  String.valueOf(maintenant.get(Calendar.SECOND)) ,10,50;
    }
}

```

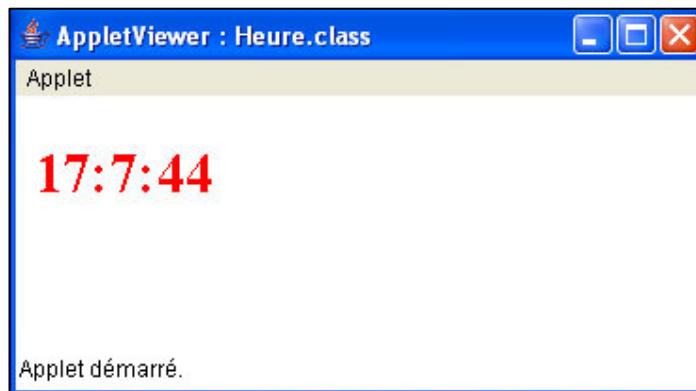
Accessoirement, on peut encore remarquer

- ♦ l'utilisation d'une couleur d'écriture particulière : on utilise pour cela la méthode

public abstract void setColor(Color c).

- ♦ le fait que l'on n'utilise pas la classe Date mais plutôt la classe **Calendar**.

On obtient cette fois bien :



Remarque

Un observateur attentif remarquera en pratique un léger tremblement de l'affichage. Ce n'est pas une illusion d'optique : ce clignotement est évidemment dû au fait que l'on réaffiche l'entièreté de l'applet à chaque seconde. Il existe des méthodes pour contourner ce problème ...

18. Les threads et les problèmes d'animations

18.1 Une animation simpliste

En soi, le principe d'une animation est simple : on affiche un élément graphique, puis on l'efface et le redessine, sous une forme un peu différente ou un peu plus loin. Nous avons pratiqué ainsi pour l'applet Heure.java affichant l'heure évoquée ci-dessus.

Nous avons aussi remarqué un **tremblement** un peu agaçant. Celui-ci est du au fait que la méthode run() appelle constamment la méthode repaint(). On se souviendra que celle-ci appelle la méthode **update()** qui *efface l'écran, redessine les contours et remplit la zone* concernée avec la couleur de fond avant d'appeler la méthode paint(). C'est cet effacement qui provoque le clignotement.

Le problème est surtout important dans le cas d'animations basées sur des images ou des graphismes relativement élaborés. Selon le type d'animation auquel on a affaire, on peut imaginer diverses solutions. Nous allons brièvement les évoquer depuis notre applet Heure.

18.2 Eliminer les clignotements : ne pas effacer l'écran

Dans le cas d'une animation résultant non pas d'un mouvement réel, mais d'un changement de couleurs donnant un effet d'ondes, on peut faire simple : puisque l'effacement d'écran dérange, supprimons-le. Il suffit pour cela de redéfinir la méthode update pour qu'elle se contente d'appeler paint :

```
public void update (Graphics g)
{
    paint(g);
}
```

Dans le cas de notre horloge, ce n'est bien entendu pas ce qu'il nous faut :

Bien clairement, il faut quand même bien effacer une partie de la zone avant d'afficher la nouvelle heure ... ☺

18.3 Éliminer les clignotements : limiter la zone à reconstruire

Une deuxième technique, pas toujours applicable, consiste à n'effacer que ce qui doit l'être, en laissant le reste de la zone figé. On utilise pour cela la méthode :

```
public abstract void setClip(int x, int y, int width, int height)
```

qui permet de définir la zone qui, seule, pourra être mise à jour par la méthode paint(). Cette dernière peut alors contenir les instructions d'"effacement" :

```
| g.setColor(Color.white);  
| g.fillRect(0,0,200, 200);
```

où l'on utilise la méthode

```
public abstract void fillRect(int x, int y, int width, int height)
```

qui remplit bien sûr la zone définie par les paramètres avec la couleur active. Il faut bien sûr comprendre qu'elle ne peut agir que dans la zone déterminée par setClip() !

```
public void paint(Graphics g)  
{  
    Font f = new Font("TimesRoman", Font.BOLD, 30);  
    g.setFont(f);  
    g.setColor(Color.white); g.fillRect(0,0,200, 200);  
    g.setColor(Color.red);  
    Date maintenant = new Date();  
    g.drawString(maintenant.getHours()+1+":"+maintenant.getMinutes()+"."  
                +maintenant.getSeconds(),5,35);  
}  
  
public void update (Graphics g)  
{  
    g.setClip(15,15,100,80); // la zone où l'on repeint  
    paint(g);  
}
```

18.4 Éliminer les clignotements : le double buffering

Dans le cas du déplacement d'images sur de grandes distances, un effet de progression lente peut être dû au fait qu'une partie de l'affichage est mis à jour avant l'autre. On peut dans ce cas **passer par une image intermédiaire**, que l'on appelle une *image virtuelle*, **sur laquelle l'affichage sera construit**. C'est ensuite cette image virtuelle qui est affichée en une seule fois sur le véritable écran :



Cette technique de l'image intermédiaire est encore, pour des raisons évidents, appelée la méthode du double buffering.

L'image virtuelle est un objet instanciant la classe **Image**, que nous déjà rencontrée dans notre étude des applets. Nous la créerons aux dimensions correspondant exactement à celles de la zone utilisée de l'applet (ou de l'image). Pour ce faire, nous utiliserons la méthode

public Dimension **size()** [JDK 1.0] ou public Dimension **getSize()** [JDK 1.1]

qui retourne une objet de la classe **Dimension**, classe dérivée directement d'Object. Elle comporte les deux variables membres publiques attendues :

public int *height*
public int *width*

L'image virtuelle proprement dite sera créée au moyen de la méthode de Component :

public Image **createImage**(int width, int height)

spécialement dédiacée à la création des images virtuelles à usage de double buffering.

Pour agir sur l'image virtuelle, nous auront besoin d'un objet Graphics. Or, la méthode de la classe Image :

public abstract Graphics **getGraphics()**

est particulièrement dédiacée à ce problème, puisqu'elle fournit un objet Graphics destiné à travailler sur une image virtuelle - cette méthode ne peut donc être utilisée que dans ce but. Une fois l'image prête, il restera à l'afficher réellement au moyen de la méthode **drawImage** déjà rencontrée précédemment. L'affichage de l'heure, avec déplacement de gauche à droite et vice versa, sera donc assurée par les modifications suivantes dans l'applet Heure :

Heure.java (2)

```
...
public class Heure extends Applet implements Runnable
{
    private volatile Thread thrHorloge;
    private Image imageVirtuelle;
    private Graphics contexteImgVirt;
    private int posX;

    public void start() { ... }

    public void init()
    {
        imageVirtuelle = createImage (size().width, size().height);
        //et pas : imageVirtuelle = new Image (size().width, size().height);
        contexteImgVirt = imageVirtuelle.getGraphics();
        posX=5;
    }

    public void run() { ... }
```

```

public void paint(Graphics g)
{
    Font f = new Font("TimesRoman", Font.BOLD, 30);
    contexteImgVirt.setFont(f);
    contexteImgVirt.setColor(Color.white);
    contexteImgVirt.fillRect(0,0,200, 200);
    contexteImgVirt.setColor(Color.red);
    Date maintenant = new Date();
    contexteImgVirt.drawString(maintenant.getHours()+1+":"+maintenant.getMinutes()
        + ":" + maintenant.getSeconds(),5,35);
    g.drawImage(image Virtuelle,posX+=5, 0, this);
    if (posX>400) posX=5;
}

public void stop() { ... }
}

```

Et voilà ...



19. Les threads et les beans

La notion de Java Bean prend de plus en plus d'importance en programmation Java (voir les volumes III et IV). Mais comment se marie-t-elle avec les threads ?

19.1 Un bean chronomètre

Supposons vouloir créer un Java Bean qui fournit un chronomètre (appelons donc **TimerBean**). Il devra évidemment savoir au bout de quelle période il devra se déclencher : une propriété est donc cette période – appelons-la **Période**¹. Il lui correspond donc une variable membre qui représente l'intervalle de temps au bout duquel le chronomètre se déclenche : appelons cette variable membre **intervalleDeclenchement**.

Notre TimerBean implémentera son chronomètre comme vu dans les paragraphes ci-dessus. Autrement dit :

- ◆ il implémente l'interface **Runnable**;
- ◆ il possède une variable membre **Thread** (disons *thrChrono*);
- ◆ ce thread est créé dans les constructeurs du bean, avec celui-ci comme paramètre, faisant ainsi du bean le target du thread - celui-ci exécutera donc, au sein de sa méthode *run()*, la méthode *run()* du bean, laquelle implémente en définitive le chronomètre.

Les utilisateurs de ce bean devront être avertis du déclenchement. pour faire court, disons qu'une deuxième propriété, **nombreTicks**, compte les déclenchements et permet de mettre en place un mécanisme de propriété liée sur cette propriété. Donc, finalement :

¹ tiens, on a déjà vu un nom de propriété de ce genre ailleurs ;-) ...

TimerBean.java

```

package Timer;

import java.util.*;
import java.beans.*;
import java.io.*;

public class TimerBean implements Runnable
{
    private Thread thrChrono;
    private int intervalleDeclenchement;
    final int intervalleDeclenchementParDefaut = 5;
    private int nombreTicks;
    protected PropertyChangeSupport GestProp;
    public TimerBean ()
    {
        intervalleDeclenchement = intervalleDeclenchementParDefaut;
        nombreTicks = 0;
        GestProp = new PropertyChangeSupport(this);
        thrChrono = new Thread(this);
    }
    public TimerBean (int iDecl)
    {
        intervalleDeclenchement = iDecl; nombreTicks = 0;
        GestProp = new PropertyChangeSupport(this);
        thrChrono = new Thread(this);
    }

    public int getPeriode() { return intervalleDeclenchement; }
    public void setPeriode(int p) { intervalleDeclenchement = p; }
    public int getNombreTicks() { return nombreTicks; }
    public void setNombreTicks (int nt)
    {
        int ancienNombreTicks = nombreTicks; nombreTicks = nt;
        System.out.println("TimerBean> changement de la propriété ");
        GestProp.firePropertyChange("nombreTicks",
            new Integer(ancienNombreTicks), new Integer(nt));
    }

    public void init() { thrChrono.start(); }

    public void run()
    {
        while (Thread.currentThread() == thrChrono && getNombreTicks() <10)
        {
            try
            {
                thrChrono.sleep(intervalleDeclenchement);
                setNombreTicks(getNombreTicks()+1);
            }
        }
    }
}

```

```
        catch (InterruptedException e) { System.out.println(e); }
    }

    public void addPropertyChangeListener(PropertyChangeListener l)
    {
        GestProp.addPropertyChangeListener(l);
    }

    public void removePropertyChangeListener(PropertyChangeListener l)
    {
        GestProp.removePropertyChangeListener(l);
    }

};
```

19.2 Le bean utilisateur : un générateur de nombres

Nous allons ensuite imaginer un générateur de nombres aléatoires qui sera avisé des changements de propriété :

GeneratorNumberBean.java

```
package Timer.Generator;

import java.beans.*;
import java.util.*;

public class GeneratorNumberBean implements PropertyChangeListener
{
    Vector nombres;

    public GeneratorNumberBean() { nombres = new Vector(); }

    public Vector getNombres() { return nombres; }

    public void propertyChange(java.beans.PropertyChangeEvent propertyChangeEvent)
    {
        Double r = new Double(Math.random()*100);
        System.out.println("GeneratorNumberBean> Un coup de timer ..." + r + " :-)");
        nombres.add(r);
    }
}
```

19.3 Le container des deux beans

Il reste à instancier et à connecter les deux beans :

TimerBeanTest.java

```
import java.beans.*;
import java.io.*;

import Timer.*;
import Timer.Generator.*;
```

```

public class TimerBeanTest
{
    public static void main (String args[])
    {
        TimerBean tb = null;
        try
        {
            tb = (TimerBean) Beans.instantiate(null,"Timer.TimerBean");
            System.out .println("main> TimerBean 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); }
        tb.setPeriode(5);
        System.out.println("main> propriété Periode initialisée");

        GeneratorNumberBean gnb = null;
        try
        {
            gnb = (GeneratorNumberBean)
                Beans.instantiate(null,"Timer.Generator.GeneratorNumberBean");
            System.out .println("main> GeneratorNumberBean 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); }

        // Connexion des deux beans
        tb.addPropertyChangeListener(gnb);
        System.out .println("main> connexion des beans réalisée");

        tb.init();
    }
}

```

Un exemple d'exécution serait :

```

main> TimerBean instancié
main> propriété Periode initialisée
main> GeneratorNumberBean instancié
main> connexion des beans réalisée
TimerBean> changement de la propriété
GeneratorNumberBean> Un coup de timer ...30.04620584045219 :-
TimerBean> changement de la propriété
GeneratorNumberBean> Un coup de timer ...54.19271084991373 :-
TimerBean> changement de la propriété
GeneratorNumberBean> Un coup de timer ...4.150031729008042 :-

```

```
TimerBean> changement de la propriété
GeneratorNumberBean> Un coup de timer ...79.63939724207174 :-)
TimerBean> changement de la propriété
GeneratorNumberBean> Un coup de timer ...38.95199540595624 :-)
TimerBean> changement de la propriété
GeneratorNumberBean> Un coup de timer ...85.09282750153248 :-)
TimerBean> changement de la propriété
GeneratorNumberBean> Un coup de timer ...94.45939235897698 :-)
TimerBean> changement de la propriété
GeneratorNumberBean> Un coup de timer ...63.10581947584944 :-)
TimerBean> changement de la propriété
GeneratorNumberBean> Un coup de timer ...30.08870018829254 :-)
TimerBean> changement de la propriété
GeneratorNumberBean> Un coup de timer ...44.40958948719877 :-)
```

19.4 Sérialiser ce qui peut l'être

Nous savons qu'un bean peut être rendu persistant: il suffit que sa classe implémente l'interface **Serializable**, ce qui n'implique la redéfinition d'aucune méthode. Donc, ici :

```
public class TimerBean implements Runnable, Serializable
{ ... }
```

Il faut cependant remarquer que notre classe comporte une variable membre *thrChrono* qui est un Thread ... et il n'est évidemment pas possible de sauvegarder un thread ! Ceci posera un problème lorsque nous tenterons de sauver un objet TimerBean avec la méthode *writeObject* : celle-ci lancera l'exception *NotSerializableException*. Il nous faut donc expliquer au compilateur que cette variable membre ne doit pas être sérialisée : ceci s'indique en Java en la qualifiant de "**transient**", ce qui prévient le compilateur qu'elle ne fait pas partie de l'objet persistant. Donc :

TimerBean.java (2)

```
public class TimerBean implements Runnable, Serializable
{
    transient private Thread thrChrono;
    private int intervalleDeclenchement; ...
}
```

Notre programme de test va donc sérialiser deux objets TimerBean dont la propriété *Periode* sera respectivement de 5 et 8 secondes. Le fichiers .ser seront placés dans le répertoire des applications.

TimerBeanSerialize.java

```
import java.io.*;
import Timer.*;
class TimerBeanSerialize
{
    private int cpt=0;
```

```

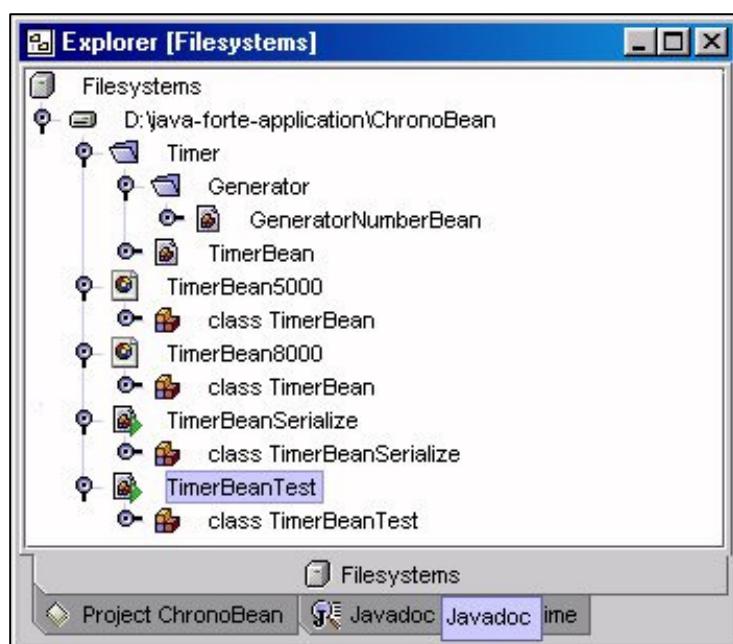
public TimerBeanSerialize ()
{
    try // Serialisations
    {
        TimerBean b = new TimerBean (5000);
        FileOutputStream fos = new FileOutputStream
            ("d:\\java-forte-application\\ChronoBean\\TimerBean5000.ser");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(b); oos.flush();

        b = new TimerBean (8000);
        fos = new FileOutputStream
            ("d:\\java-forte-application\\ChronoBean\\TimerBean8000.ser");
        oos = new ObjectOutputStream(fos);
        oos.writeObject(b); oos.flush();
        oos.close();
    }
    catch (NotSerializableException e)
    { System.out.println("Composante non sérialisable -- " + e); }
    catch (IOException e)
    { System.out.println(e); }
}

public static void main(String[] args)
{
    TimerBeanSerialize tbt = new TimerBeanSerialize ();
}
}

```

Au sein d'un environnement comme Netbeans, par exemple, on peut remarquer que l'EDI a reconnu le bean sérialisé dans le répertoire monté :



Cependant, l'exécution du programme de test qui tente de retrouver un bean sérialisé rencontre quelques problèmes, soit une NullPointerException :

```
main> TimerBean instancié
main> propriété Periode initialisée
jmaavian.>l aGnegr.eNrualtloProNiunmbeerrEBxecaenp tiinosnt
an caité
Timmaeirn.>T icmoenrnBeexaino.ni ndiets( TbiemaenrsB eraéna.ljasvéae:
44)
    at TimerBeanTest.main(TimerBeanTest.java:72)
Exception in thread "main"
```

Groups ??? Un peu de réflexion nous fait comprendre le problème : l'instanciation du thread, non sérialisé, n'a pu se faire puisqu'elle est programmée dans le constructeur ... qui n'a pas été ici appelé ! Il nous faut donc déplacer cette instanciation dans la méthode init() :

TimerBean.java (3)

```
package Timer;

import java.util.*;
import java.beans.*;
import java.io.*;

public class TimerBean implements Runnable, Serializable
{
    transient private Thread thrChrono;
    private int intervalleDeclenchement;
    final int intervalleDeclenchementParDefaut = 5;
    private int nombreTicks;

    protected PropertyChangeSupport GestProp;

    public TimerBean ()
    {
        intervalleDeclenchement = intervalleDeclenchementParDefaut;
        GestProp = new PropertyChangeSupport(this);
        // thrChrono = new Thread(this); - SUPPRIME
    }
    public TimerBean (int iDecl)
    {
        intervalleDeclenchement = iDecl;
        GestProp = new PropertyChangeSupport(this);
        // thrChrono = new Thread(this); - SUPPRIME
    }

    public int getPeriode() { return intervalleDeclenchement; }

    ...
}
```

```
public void init()
{
    thrChrono = new Thread(this);
    // DEPLACE ICI pour les besoins d'un bean sérialisé
    thrChrono.start();
}
...
};
```

Tout se passe alors nettement mieux :

```
main> TimerBean instancié
main> propriété Periode initialisée
main> GeneratorNumberBean instancié
main> connexion des beans réalisée
TimerBean> changement de la propriété
GeneratorNumberBean> Un coup de timer ...70.7229779428076 :-
TimerBean> changement de la propriété
GeneratorNumberBean> Un coup de timer ...55.672158511009314 :-
TimerBean> changement de la propriété
...
TimerBean> changement de la propriété
GeneratorNumberBean> Un coup de timer ...81.88285130219458 :-)
```

20. Un outil multithreadé pour Java Beans : la BeanBox

20.1 Un outil de développement des beans

Sun fournit depuis le début de Java un environnement de test et de configuration des beans appelé le **BDK** (**B**eans **D**evelopment **K**it). On y trouve essentiellement un container de référence appelé la "**BeanBox**" : il s'agit d'un interface graphique à 3 fenêtres destiné à permettre le test et la configuration de beans préalablement écrits. Le BDK fournit également la source d'un certain nombre de beans prêts à l'emploi, comme le célèbre Duke jongleur dont l'effigie orne l'entête de ce chapitre.

L'outil est à présent obsolète. Utilisons-le cependant à titre d'exemple. Ouvrons une fenêtre DOS et plaçons nous dans le répertoire c:\BDK10\beanbox. Après avoir complété le path courant par le chemin permettant de trouver l'interpréteur Java :

```
| c:\BDK10\beanbox> set path= C:\j2sdk1.4.2_03\bin;%path%
```

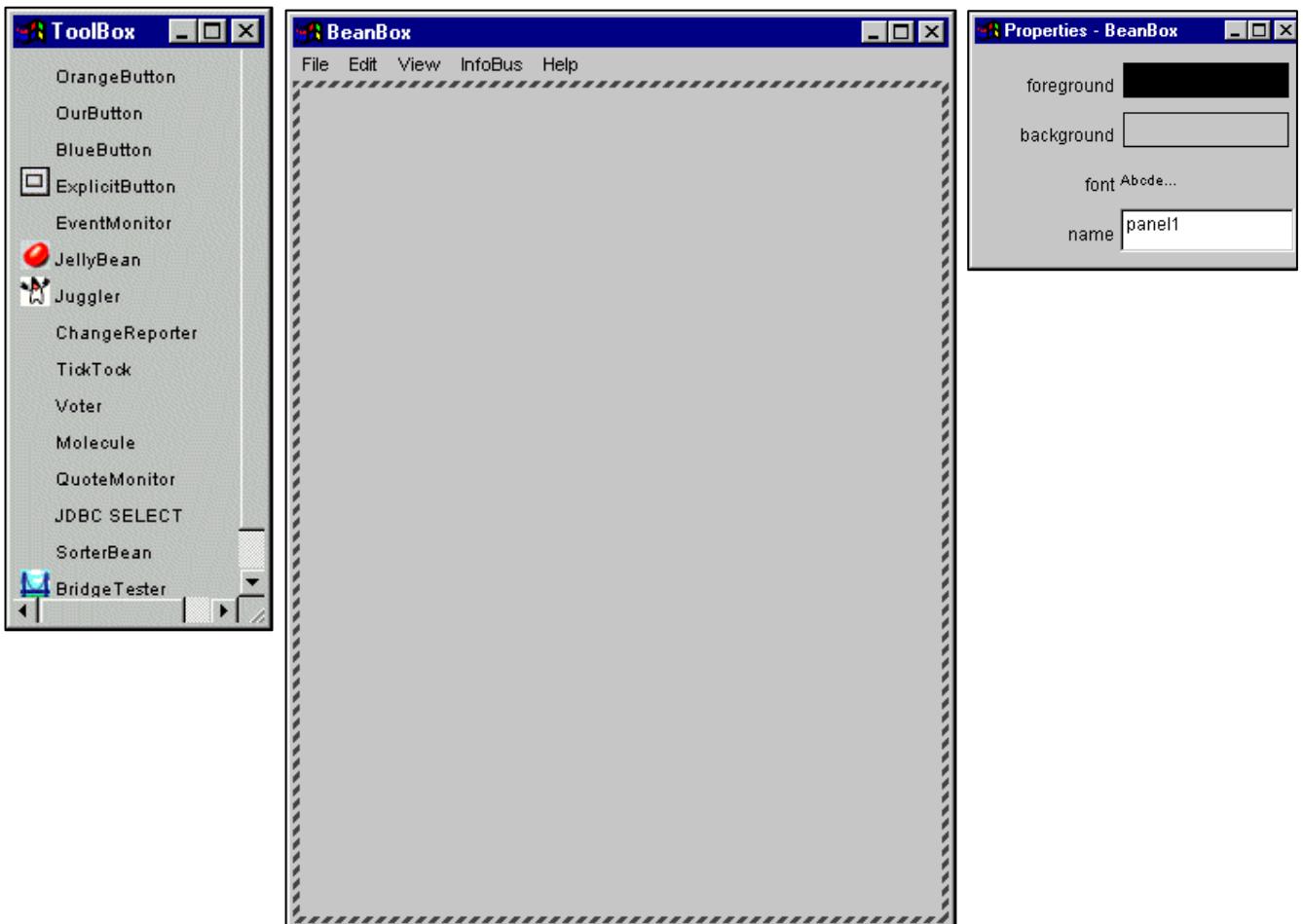
on peut lancer la beanbox par un simple run sur la ligne de commande :

```
| c:\BDK10\beanbox> run
```

Un fichier bat est en fait exécuté :

```
C:\BDK10\beanbox>if "" == "Windows_NT" setlocal
C:\BDK10\beanbox>set CLASSPATH=classes;JavaScope.zip;infobus.jar
C:\BDK10\beanbox>java sun.beanbox.BeanBoxFrame
```

L'interpréteur Java s'est donc mis au travail. Après le passage furtif d'une boîte de dialogue annonçant la recherche et l'analyse de fichiers jar, on voit apparaître trois fenêtres java :

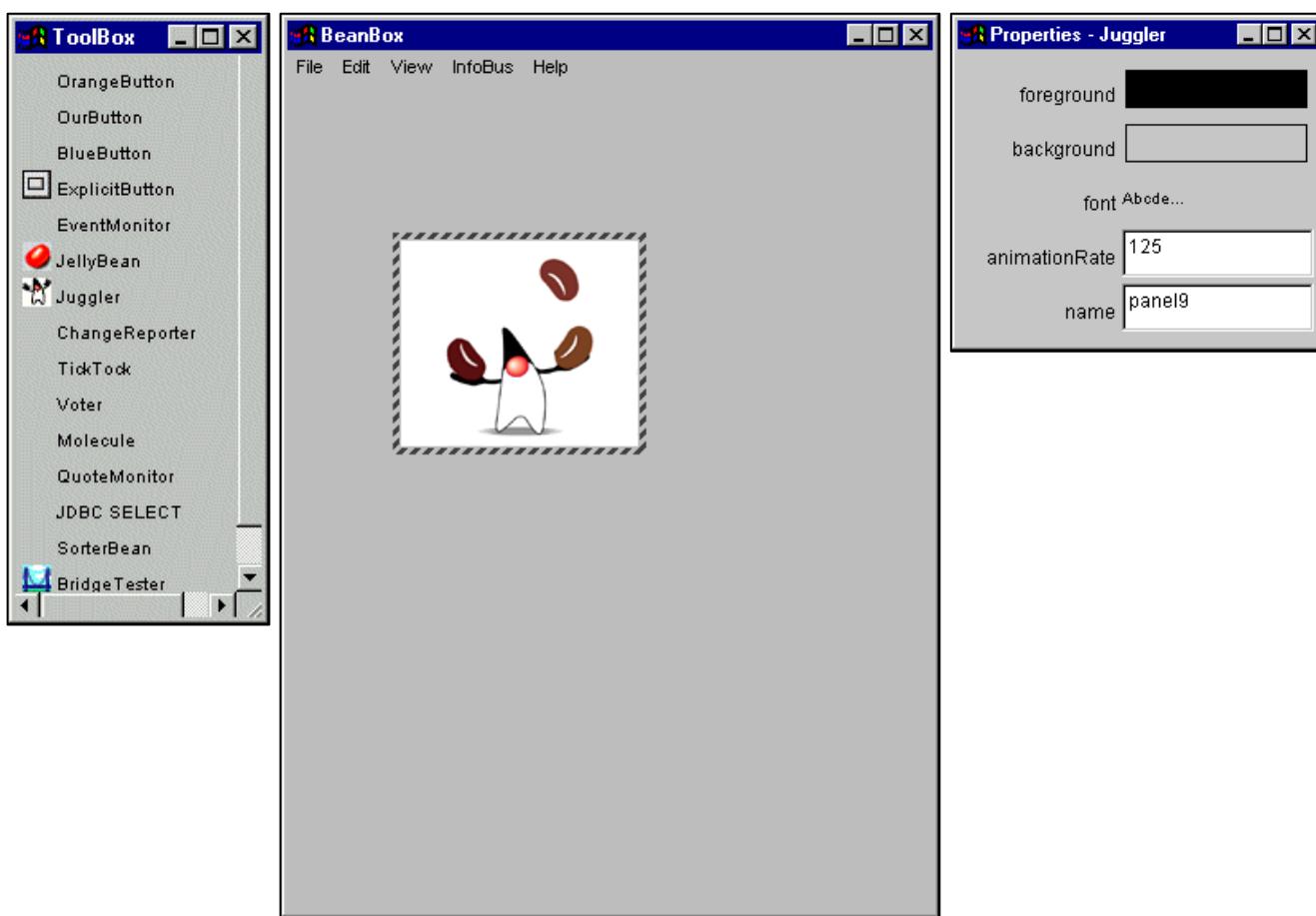


- 1) au centre, la fenêtre **Beanbox** proprement dite : elle constitue le véritable container de test des beans et propose d'ailleurs un menu pour réaliser cette tâche;
- 2) à gauche, la fenêtre **Toolbox** : elle propose une liste de beans disponibles (comme notre jongleur : *Juggler*);
- 3) à droite, la fenêtre **Properties** : elle affiche les propriétés du bean sélectionné, ou de la Beanbox si il n'y en a pas; selon le bean, cette fenêtre des propriétés se modifiera pour s'adapter aux propriétés.

On peut encore remarquer que le programme de lancement reste en attente dans la fenêtre DOS et affiche des informations sur le déroulement des opérations ou les erreurs.

20.2 Placer un bean prédéfini dans la beanbox

Placer un bean dans la Beanbox n'est pas trop difficile : on le sélectionne en cliquant dans la Toolbox (le curseur prend alors la forme d'une croix) après quoi on clique une deuxième fois dans la beanbox. Le bean apparaît alors tandis que la fenêtre des properties s'adapte automatiquement et affiche les propriétés correspondantes. Ainsi, pour notre jongleur, le triptyque devient :



L'objectif est de pouvoir ainsi paramétriser le bean depuis la fenêtre des propriétés et de sauver ainsi son nouvel état. Ainsi, si nous souhaitons que notre jongleur joue plus rapidement, nous allons modifier le champ animationRate en le portant à 50. Bien clairement, la face cachée de la beanbox comporte des threads, des événements et des pipes ...

21. Les synchroniseurs du JDK 1.5

Depuis sa version 1.5, le JSR propose de nouvelles classes permettant de mettre en place l'un ou l'autre type de synchronisation. On peut ainsi trouver dans le nouveau package `java.util.concurrent` :

- des compteurs d'événements ou loquets (**CountDownLatch**);
- des barrières cycliques (**CyclicBarrier**) qui permettent de synchroniser des threads coopératifs (donc qui s'attendent les uns les autres); en fait, ce sont des compteurs d'événements réinitialisables;
- des échangeurs (**Exchanger** - c'est une classe template), qui ne font rien d'autre que de synchroniser deux threads avec échanges de données au point de synchronisation au moyen de pipes encapsulés);
- et surtout des sémaphores (**Semaphore**) !

21.1 Les sémaphores

Nous retrouvons donc ici ces vieilles connaissances de la théorie des systèmes d'exploitation. Pour rappel, un sémaphore sert à mémoriser le nombre de processus ou de threads en sommeil, c'est-à-dire en attente d'un événement. Son inventeur, Dijkstra,

l'accompagna de deux opérations utilisables par les processus ou les threads voulant accéder à une ressource :

- ◆ **down** : si le sémaphore est à 0 (rien n'est disponible), le processus est mis en sommeil; si il est positif, le sémaphore est décrémenté et le processus accède;
- ◆ **up** : le sémaphore est incrémenté; si des processus sont en sommeil sur le sémaphore, l'un d'entre eux est réveillé pour réaliser le down qui l'a bloqué.

On peut donc encore considérer que la valeur d'un sémaphore est le nombre de *permis* ou de *jetons* d'accès à un ressource (dans le cas particulier où cette valeur ne peut dépasser 1, on parle de sémaphore binaire ou mutex).

Java propose donc, depuis le JDK 1.5, la classe **Semaphore** à l'usage des treads. Les deux constructeurs

```
public Semaphore(int permits)
public Semaphore(int permits, boolean fair)
```

permettent de fixer le nombre initial de permis disponibles; l'éventuel 2^{ème} paramètre permet de déterminer si ils doivent être gérés en FIFO ou pas.

Les deux méthodes importantes sont :

```
public void acquire()throws InterruptedException
```

Il s'agit du down : le thread tente d'acquérir un permis d'accès. Si sa tentative est couronnée de succès, il accède aux ressources et le nombre de permis disponibles est diminué d'une unité. Sinon, il est mis en sommeil, en fait en attente d'une éventuelle interruption (cas peu intéressant) mais surtout de l'invocation par un autre thread de la méthode :

```
public void release()
```

Il s'agit du up : un thread a terminé ses accès aux ressources et rend un permis disponible au sémaphore (la valeur de celui-ci est donc incrémentée). Un thread est choisi parmi ceux en attente (si il y en a) pour effectuer son down (donc se débloquer de sa méthode acquire()).

Nous allons utiliser cette classe dans le contexte, classique dans la théorie des systèmes d'exploitation, "des coiffeurs et des clients" – ici, nous dirons plutôt que des threads producteurs produisent des ressources qu'ils placent dans un stock-containier (où la place disponible est limitée) tandis que des threads utilisateurs avides attendent le moment propice pour traiter les ressources produites.

Bien clairement,

- ◆ un producteur ne peut déposer une ressource produite dans le stock que si il y a au moins une place libre – un sémaphore "*libre*" sera donc chargé de gérer cet aspect;
- ◆ un utilisateur ne devrait pouvoir prendre une ressource que si une ressource au moins est disponible – un second sémaphore "*disponible*" sera chargé de gérer cet autre aspect.

La première classe à écrire est donc celle qui représente le stock de ressources : elle gère une liste liée contenant les ressources proprement dites (ici, de simples chaîne de caractères) et les deux sémaphores évoqués. Bien sûr, l'accès à cette liste (que ce soit pour placer ou pour prendre) doit être atomique : les deux opérations de type get() et put() seront des méthodes synchronized !

Ressources.java

```
/*
 * Ressources.java
 *
 */
package threadsnew;

/**
 * @author Vilvens
 */

import java.util.concurrent.*;
import java.util.*;

public class Ressources
{
    private int nbreMaxRessources;
    private final Semaphore disponible, libre;
    private LinkedList stockRessources;

    public Ressources(int n) throws InterruptedException
    {
        nbreMaxRessources = n;
        disponible = new Semaphore(nbreMaxRessources, true);
        disponible.acquire(nbreMaxRessources) ;
        libre = new Semaphore(nbreMaxRessources, true);
        stockRessources = new LinkedList();
    }

    public Object getRessource() throws InterruptedException
    {
        disponible.acquire();
        Object r = getRessourceSuivante();
        libre.release();
        return r;
    }

    public void putRessource(Object x) throws InterruptedException
    {
        System.out.println("-- tentative de placement de " + x);
        libre.acquire();
        putNouvelleRessource(x);
        disponible.release();
    }

    protected synchronized Object getRessourceSuivante()
    {
        return stockRessources.remove();
    }
}
```

```
protected synchronized void putNouvelleRessource(Object r)
{
    stockRessources.addLast(r);
}
```

On remarquera encore que le sémaphore disponible commence par prendre tous les permis afin de bloquer les threads utilisateurs dès leur démarrage (sinon, ils accèderaient une liste vide).

Les producteurs et utilisateurs s'écrivent alors sans trop de problèmes :

ProducteurRessource.java

```
/*
 * ProducteurRessource.java
 *
 */
package threadsnew;
/**
 * @author Vilvens
 */

public class ProducteurRessource extends Thread
{
    private Ressources stock;
    private String nomP;

    public ProducteurRessource (Ressources t, String n) { stock = t; nomP = n; }

    public void run()
    {
        int nbreRessourcesProduites = (int) (Math.random()*10) + 4;
        System.out.println("Le producteur " + nomP + " va produire " +
                           nbreRessourcesProduites + " ressources");
        for (int i=0; i <nbreRessourcesProduites; i++)
        {
            String ressourceFabriquee = String.valueOf((int) (Math.random()*50000));
            try
            {
                System.out.println("Le producteur " + nomP + " dépose " +
                                   ressourceFabriquee +" dans le stock");
                stock.putRessource(ressourceFabriquee);
                int temps = (int) (Math.random()*Atelier.TEMPS_PRODUCTION);
                sleep(temps);
            }
            catch (InterruptedException e) {}
        }
        System.out.println("*** Le producteur " + nomP + " a terminé");
    }
}
```

UtilisateurRessource.java

```
/*
 * UtilisateurRessource.java
 *
 */
package threadsnew;

/**
 * @author Vilvens
 */

public class UtilisateurRessource extends Thread
{
    private Ressources stock;
    private String nomU;

    public UtilisateurRessource (Ressources t, String n) { stock = t; nomU = n; }

    public void rungetRessource();
                System.out.println("L'utilisateur " + nomU + " retire du stock : " +
                                   ressourceFabriquee);
                int temps = (int) (Math.random() * Atelier.TEMPS_UTILISATION);
                sleep(temps);
            }
            catch (InterruptedException e) {}
        }
    }
}
```

Il ne reste plus qu'à instancier ces différentes classes au sein d'une classe Atelier (qui contient donc TEMPS_PRODUCTION et TEMPS_UTILISATION) :

Atelier.java

```
/*
 * Atelier.java
 *
 */
package threadsnew;

/**
 * @author Vilvens
 */


```

```
public class Atelier
{
    private static final int NBRE_PLACES_STOCK = 3;
    private static final int NBRE_PRODUCTEURS = 2;
    private static final int NBRE_UTILISATEURS = 5;
    public static final int TEMPS_PRODUCTION = 1000;
    public static final int TEMPS_UTILISATION = 15000;

    public static void main(String[] args)
    {
        Ressources LeStock = new Ressources (NBRE_PLACES_STOCK);
        UtilisateurRessource thrUtilisa[] =
            new UtilisateurRessource[NBRE_UTILISATEURS];
        for (int i=0; i<NBRE_UTILISATEURS; i++)
        {
            thrUtilisa[i] = new UtilisateurRessource(LeStock, "utilisateur"+String.valueOf(i));
            thrUtilisa[i].start();
        }
        ProducteurRessource thrProd[] =
            new ProducteurRessource[NBRE_PRODUCTEURS];
        for (int i=0; i<NBRE_PRODUCTEURS; i++)
        {
            thrProd[i] = new ProducteurRessource(LeStock, "producteur"+String.valueOf(i));
            thrProd[i].start();
        }
    }
}
```

Un exemple d'exécution serait :

```
Le producteur producteur0 va produire 13 ressources
Le producteur producteur0 dépose 21777 dans le stock
-- tentative de placement de 21777
    L'utilisateur utilisateur0 retire du stock : 21777
Le producteur producteur1 va produire 11 ressources
Le producteur producteur1 dépose 4367 dans le stock
-- tentative de placement de 4367
    L'utilisateur utilisateur1 retire du stock : 4367
Le producteur producteur1 dépose 15671 dans le stock
-- tentative de placement de 15671
    L'utilisateur utilisateur2 retire du stock : 15671
Le producteur producteur0 dépose 41659 dans le stock
-- tentative de placement de 41659
    L'utilisateur utilisateur3 retire du stock : 41659
Le producteur producteur0 dépose 46366 dans le stock
-- tentative de placement de 46366
    L'utilisateur utilisateur4 retire du stock : 46366
Le producteur producteur1 dépose 7788 dans le stock
-- tentative de placement de 7788
```

```
Le producteur producteur1 dépose 31889 dans le stock
-- tentative de placement de 31889
Le producteur producteur1 dépose 31611 dans le stock
-- tentative de placement de 31611
Le producteur producteur0 dépose 9695 dans le stock
-- tentative de placement de 9695
Le producteur producteur1 dépose 28114 dans le stock
-- tentative de placement de 28114
    L'utilisateur utilisateur1 retire du stock : 7788
Le producteur producteur0 dépose 34845 dans le stock
-- tentative de placement de 34845
    L'utilisateur utilisateur0 retire du stock : 31889
Le producteur producteur1 dépose 12131 dans le stock
-- tentative de placement de 12131
    L'utilisateur utilisateur1 retire du stock : 31611
Le producteur producteur0 dépose 34611 dans le stock
-- tentative de placement de 34611
    L'utilisateur utilisateur3 retire du stock : 9695
    L'utilisateur utilisateur3 retire du stock : 28114
Le producteur producteur0 dépose 1971 dans le stock
-- tentative de placement de 1971
Le producteur producteur1 dépose 32327 dans le stock
-- tentative de placement de 32327
    L'utilisateur utilisateur4 retire du stock : 34845
Le producteur producteur0 dépose 37750 dans le stock
-- tentative de placement de 37750
...
Le producteur producteur1 dépose 33023 dans le stock
-- tentative de placement de 33023
    L'utilisateur utilisateur4 retire du stock : 15308
Le producteur producteur0 dépose 20804 dans le stock
-- tentative de placement de 20804
    L'utilisateur utilisateur2 retire du stock : 2006
*** Le producteur producteur1 a terminé
    L'utilisateur utilisateur3 retire du stock : 2620
Le producteur producteur0 dépose 3968 dans le stock
-- tentative de placement de 3968
    L'utilisateur utilisateur1 retire du stock : 48382
Le producteur producteur0 dépose 44408 dans le stock
-- tentative de placement de 44408
    L'utilisateur utilisateur0 retire du stock : 33023
*** Le producteur producteur0 a terminé
    L'utilisateur utilisateur4 retire du stock : 20804
    L'utilisateur utilisateur3 retire du stock : 3968
    L'utilisateur utilisateur2 retire du stock : 44408
```

21.2 Les compteurs d'événements

Evidemment, on pourra objecter que l'application ne se termine pas lorsque tous les threads ont effectué leur travail. Nous allons corriger cela en utilisant un compteur d'événements **CountDownLatch**. Un tel objet se construit avec le nombre total d'événements dont il faut surveiller la réalisation :

```
public CountDownLatch(int count)
```

Chaque fois qu'un thread a terminé un tel événement, il peut décrémenter le compteur avec :

```
public void countDown()
```

Quant au thread qui attend la réalisation de tous ces événements, il utilisera :

```
public void await() throws InterruptedException
```

qui bloque jusqu'à ce que le compteur tombe à 0.

Pour notre application, nous allons conditionner la terminaison du thread principal à un tel compteur : il va permettre d'attendre la fin de toutes les opérations. Le nombre de celles-ci est donc le nombre de ressources produites multiplié par 2. Les producteurs et utilisateurs se verront passer ce compteur pour le décrémenter au fur et à mesure de leurs activités.

Atelier.java (2)

```
/*
 * Atelier.java
 */
package threadsnew;

/**
 * @author Vilvens
 */
import java.util.concurrent.*;

public class Atelier
{
    private static final int NBRE_PLACES_STOCK = 3;
    private static final int NBRE_PRODUCTEURS = 2;
    private static final int NBRE_UTILISATEURS = 5;
    public static final int TEMPS_PRODUCTION = 1000;
    public static final int TEMPS_UTILISATION = 15000;

    public static void main(String[] args) throws InterruptedException
    {
        Ressources LeStock = new Ressources (NBRE_PLACES_STOCK);
        int nbreOperations =0;
```

```

ProducteurRessource thrProd[] = new ProducteurRessource[NBRE_PRODUCTEURS];
for (int i=0; i<NBRE_PRODUCTEURS; i++)
{
    thrProd[i] = new ProducteurRessource(LeStock, "producteur"+String.valueOf(i));
    nbreOperations += thrProd[i].getNbreRessourcesProduites();
}

CountDownLatch cpt = new CountDownLatch(nbreOperations*2);

for (int i=0; i<NBRE_PRODUCTEURS; i++) thrProd[i].setCompteurOperations(cpt);

UtilisateurRessource thrUtilisa[] =
    new UtilisateurRessource[NBRE_UTILISATEURS];
for (int i=0; i<NBRE_UTILISATEURS; i++)
{
    thrUtilisa[i] =
        new UtilisateurRessource(LeStock, "utilisateur"+String.valueOf(i), cpt);
}

for (int i=0; i<NBRE_UTILISATEURS; i++) thrUtilisa[i].start();
for (int i=0; i<NBRE_PRODUCTEURS; i++) thrProd[i].start();

try
{
    cpt.await();
}
catch( InterruptedException e) {}
System.out.println("L'atelier a terminé ses fabrications");
System.exit(0);
}
}
}

```

Sans surprise, voici les rectifications dans le producteur et l'utilisateur :

ProducteurRessource.java (2)

```

/*
 * ProducteurRessource.java
 */
package threadsnew;

/**
 * @author Vilvens
 */

import java.util.concurrent.*;

public class ProducteurRessource extends Thread
{
    private Ressources stock;

```

```

private String nomP;
private CountDownLatch cptOperations;
private int nbreRessourcesProduites;

public ProducteurRessource (Ressources t, String n)
{
    stock = t;
    nomP = n;
    nbreRessourcesProduites = (int) (Math.random()*10) + 4;
}

public int getNbreRessourcesProduites() { return nbreRessourcesProduites; }
public void setCompteurOperations(CountDownLatch c)
{
    cptOperations = c;
}

public void run()
{
    System.out.println("Le producteur " + nomP + " va produire " +
        nbreRessourcesProduites + " ressources");
    for (int i=0; i <nbreRessourcesProduites; i++)
    {
        String ressourceFabriquee = String.valueOf((int) (Math.random()*50000));
        try
        {
            System.out.println("Le producteur " + nomP + " dépose " +
                ressourceFabriquee + " dans le stock");
            stock.putRessource(ressourceFabriquee);
            int temps = (int) (Math.random()*Atelier.TEMPS_PRODUCTION);
            sleep(temps);
            System.out.println(" P>> " + cptOperations.getCount());
            cptOperations.countDown();
        }
        catch (InterruptedException e) {}
    }
    System.out.println("*** Le producteur " + nomP + " a terminé");
}
}

```

UtilisateurRessource.java

```

/*
 * UtilisateurRessource.java
 */
package threadsnew;

/**
 * @author Vilvens
 */

```

```
import java.util.concurrent.*;  
  
public class UtilisateurRessource extends Thread  
{  
    private Ressources stock;  
    private String nomU;  
    private CountDownLatch cptOperations;  
  
    public UtilisateurRessource (Ressources t, String n, CountDownLatch c)  
    {  
        stock = t;  
        nomU = n;  
        cptOperations = c;  
    }  
  
    public void run()  
    {  
        while(true )  
        {  
            try  
            {  
                String ressourceFabriquee = (String)stock.getRessource();  
                System.out.println("L'utilisateur " + nomU + " retire du stock : " +  
                    ressourceFabriquee);  
                int temps = (int) (Math.random()*Atelier.TEMPS_UTILISATION);  
                sleep(temps);  
                System.out.println(" U>> " + cptOperations.getCount());  
                cptOperations.countDown();  
            }  
            catch (InterruptedException e) {}  
        }  
    }  
}
```

Un exemple d'exécution sera :

Le producteur producteur0 va produire 9 ressources

Le producteur producteur0 dépose 39105 dans le stock
-- tentative de placement de 39105

L'utilisateur utilisateur0 retire du stock : 39105

Le producteur producteur1 va produire 13 ressources

Le producteur producteur1 dépose 16620 dans le stock
-- tentative de placement de 16620

L'utilisateur utilisateur1 retire du stock : 16620

P>> 44

Le producteur producteur0 dépose 16470 dans le stock
-- tentative de placement de 16470

L'utilisateur utilisateur2 retire du stock : 16470

P>> 43

...

U>> 36

L'utilisateur utilisateur4 retire du stock : 44980

P>> 35

Le producteur producteur1 dépose 5338 dans le stock

-- tentative de placement de 5338

...

***** Le producteur producteur0 a terminé**

U>> 12

L'utilisateur utilisateur2 retire du stock : 5330

...

***** Le producteur producteur1 a terminé**

U>> 8

L'utilisateur utilisateur0 retire du stock : 45228

U>> 7

L'utilisateur utilisateur3 retire du stock : 503

U>> 6

L'utilisateur utilisateur2 retire du stock : 10942

U>> 5

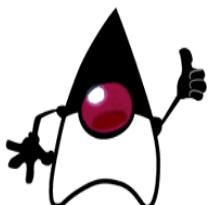
U>> 4

U>> 3

U>> 2

U>> 1

L'atelier a terminé ses fabrications



Le succès actuel de Java trouve sa principale source dans Internet.
Donc, il s'agirait de programmer des communications réseaux.
Commençons par les paradigmes classiques : flux et sockets, nous voici !

XI. Les communications réseaux



Internet fut un cadeau de Dieu

(K. Polese, responsable des produits Java)

Nous allons revenir aux flux ! En effet, le langage Java fournit au moins autant de support pour les E/S réseaux que pour les E/S fichiers. De plus, nous allons constater que les communications réseau sont gérées en Java au moyen de flux primaires du type `InputStream/OutputStream` sur lesquels on peut alors greffer des flux de plus haut niveau.

Mais prenons les choses dans l'ordre² ...

1. Les protocoles d'Internet

Faut-il encore le dire ? Les machines qui communiquent sur Internet le font en utilisant TCP/IP. Plus précisément, le modèle suivi comporte 4 couches :

Application	HTTP, FTP, Telnet,
Transport	TCP, UDP, ...
Réseau	IP,
Physique	Ethernet

TCP (Transport Control Protocol) est un protocole orienté connexion fournissant un transport fiable de données entre deux machines : les données sont reçues dans le même ordre que celui selon lequel elles ont été émises, sans quoi une erreur est détectée.

L'assurance que l'ordre est respecté est critique pour des applications utilisant **HTTP**, protocole applicatif (basé TCP) du Web par excellence . La vérification diminue évidemment les performances. Par contre, certaines applications n'ont pas besoin de cette contrainte, comme par exemple les requêtes SNMP. Dans ce cas, on peut utiliser **UDP** (User Datagram Protocol) qui est un protocole non orienté connexion fournissant un transport non fiable de données entre deux machines : les données sont transmises par paquets, appelés des **datagrammes**, sans garantie quant à l'arrivée effective de ces données. L'ordre des paquets n'est pas important et chacun de ceux-ci est indépendant des autres.

Java fournit diverses classes permettant de communiquer au moyen de ces protocoles, donc en particulier sur Internet. Ces classes se trouvent dans le package **java.net**. Citons d'ores et déjà :

- ◆ Socket et SocketServer : pour le protocole TCP;
- ◆ DatagramPacket et DatagramServer : pour le protocole UDP;
- ◆ URL et URLConnection : pour le protocole HTTP.

² les 2 paragraphes qui suivent s'adressent aux lecteurs qui n'auraient aucune connaissance de TCP/IP – donc, les autres peuvent passer au paragraphe 3 ...

2. Les ports

En général, un ordinateur possède une seule connexion au réseau. Cependant, il peut arriver que des applications différentes utilisent la même connexion. Comment, dès lors, savoir à quelle application le paquet arrivant est destiné ? En utilisant des ports.

Le numéro de port, codé sur 16 bits, accompagne l'adresse IP, codée sur 32 bits, qui désigne le destinataire des données. Plus précisément :

- ◆ en mode connexion, une application établit une connexion en liant une socket à un numéro de port; évidemment, deux applications ne peuvent se lier au même port.
- ◆ en mode datagramme, le paquet est accompagné du numéro de port.

Un numéro de port est donc un nombre positif dont la valeur se trouve dans l'intervalle [0,65535]. Cependant, les numéros de l'intervalle [0,1023] sont réservés à des services bien connus des protocoles http et ftp [*well known ports*]. Ainsi :

- ◆ HTTP utilise le port 80;
- ◆ FTP le port 21;
- ◆ Telnet le port 23.

3. Une communication client-serveur basée TCP en mode console

On peut travailler en Java au niveau transport. Un serveur est une application qui est "à l'écoute" sur un port précis, attendant qu'un client demande une connexion. Lorsque c'est le cas, le serveur et le client établissent une connexion dédiée à leurs échanges. Chacun attache à un port une "**socket**" : on désigne ainsi un point terminal d'une communication entre deux programmes tournant sur le réseau. Chacun des deux interlocuteurs utilisera sa socket pour y lire et y écrire à la destination de l'autre. A remarquer que le serveur s'est alloué pour cela un autre port, restant ainsi à l'écoute sur le port d'écoute.

La classe **Socket** implémente une socket de type client; la classe **ServerSocket** implémente bien sûr une socket de type serveur.

3.1 La classe Socket : le client

Elle possède de nombreux constructeurs, dont nous retiendrons :

```
public Socket(InetAddress address, int port) throws IOException // adresse IP et port  
public Socket(String host, int port) throws UnknownHostException, IOException  
           // nom (au sens large) et port
```

Il est possible d'associer des flux d'entrée et de sortie à la socket au moyen des méthodes :

```
public OutputStream getOutputStream() throws IOException  
public InputStream getInputStream() throws IOException
```

On pourra ainsi créer sur ces flux des objets DataInputStream et DataOutputStream (ou équivalents) permettant la communication.

Dans le programme suivant, un client se connecte sur un serveur situé sur une machine dont on fournit l'adresse IP (son nom aurait pu convenir aussi); le serveur est supposé écouter sur le port 50000 – nous verrons comment dans le paragraphe suivant. Ce serveur gère des

portefeuilles d'actions. Le client lui envoie le nom d'un actionnaire et une quantité d'actions à vendre. Le serveur recherche ce nom et décrémente le portefeuille si possible.

ClientSocket.java

```
/*
 * ClientSocket.java
 */

package clientserveursocket;

/**
 * @author Vilvens
 */

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

public class ClientSocket
{
    public static void main(String[] args)
    {
        Socket cliSock = null;
        DataInputStream dis = null;
        DataOutputStream dos = null;
        try
        {
            cliSock = new Socket("192.168.1.8", 50000);
            System.out.println("Client connecté : " + cliSock.getInetAddress().toString());
        }
        catch (UnknownHostException e)
        {
            System.err.println("Erreur ! Host non trouvé [" + e + "]");
        }
        catch (IOException e)
        {
            System.err.println("Erreur ! Pas de connexion ? [" + e + "]");
        }

        try
        {
            dis = new DataInputStream(cliSock.getInputStream());
            dos = new DataOutputStream(cliSock.getOutputStream());
            System.out.println("Flux créés");

            if (cliSock==null || dis==null || dos==null) System.exit(1);

            BufferedReader disClavier = new BufferedReader (
                new InputStreamReader (System.in));
            outNom = disClavier.readLine();
            System.out.println("outNom =" + outNom);
            outQuantite = Integer.parseInt(disClavier.readLine());
            System.out.println("outQuantite =" + outQuantite);
            disClavier.close();
        }
    }
}
```

```
catch (IOException e)
    { System.err.println("Erreur ! Pas de connexion ? [" + e + "]"); }

String reponse = null;
int inQuantiteRestante = 0;
System.out.println("outNom = " + outNom + " et outQuantite =" + outQuantite);
if (!outNom.equals("") && outQuantite>0 && dos!=null && dis!=null)
{
    System.out.println("Client au travail");
    try
    {
        dos.writeUTF(outNom);
        dos.writeInt(outQuantite);
        reponse = dis.readUTF();
        inQuantiteRestante = dis.readInt();
        System.out.println("Reponse obtenue = " + reponse + " : quantité restante = " +
                           inQuantiteRestante);
        dis.close(); dos.close(); cliSock.close();
    }
    catch (IOException e)
    {
        System.err.println("Erreur ! Pas de connexion ? [" + e + "]");
    }
}
}
```

On remarquera

- ◆ les écritures sur le réseau avec les méthodes appropriées aux données de **DataOutputStream** (`writeUTF()` et `writeInt()`);
- ◆ les lectures sur ce même réseau avec les méthodes tout aussi appropriées aux données de **DataInputStream** (`readUTF()` et `readInt()`);
- ◆ les lectures clavier (rare en Java !) au moyen des flux **BufferedReader** et **InputStreamReader**.

L'exécution donne :

```
Client connecté : /192.168.1.8
Flux créés
Mercenier
outNom =Mercenier
25
outQuantite =25
outNom = Mercenier et outQuantite = 25
Client au travail
Reponse obtenue = Mercenier : quantité restante = 15205
```

On remarquera aussi l'utilisation de la méthode de la classe `Socket`

```
public InetAddress getInetAddress()
```

qui fournit un objet instance d'une classe héritée de la classe **InetAddress** représentant une adresse IP et dont les méthodes permettent d'obtenir les informations attachées à cette adresse. En fait, si on glisse dans le code l'instruction supplémentaire :

```
System.out.println("Classe InetAddress : " + cliSock.getInetAddress().getClass().getName());
```

on obtient dans la console :

```
| Classe InetAddress : java.net.Inet4Address
```

Il s'agit bien sûr d'une classe dérivée d'InetAddress, qui possède une sœur (**Inet6Address**) : c'est bien ça, la plate-forme Java s'adapte à un stack TCP/IP4 ou TCP/IP6 ☺.

Voyons à présent comment le serveur a construit sa réponse.

3.2 La classe ServerSocket : le serveur

Elle permet la réalisation du serveur d'une connexion client/serveur par socket. La classe possède plusieurs constructeurs, dont nous retiendrons :

```
public ServerSocket(int port) throws IOException
```

qui permet de spécifier sur quel port le serveur attend. Si le serveur sait effectivement se connecter sur le port désigné, un objet ServerSocket est créé et est mis à l'écoute (*listen*) sur ce port. Le serveur peut alors se mettre en attente d'une connexion client au moyen de la méthode :

```
public Socket accept() throws IOException
```

L'appel de cette méthode est bloquant jusqu'à qu'une connexion soit effectuée. Un maximum de 50 connexions est prévu. La méthode renvoie la socket correspondant à la connexion établie; elle est liée à un autre port client.

Voici le serveur correspondant au client ci-dessus. Ses données sont contenues dans une Hashtable statique (en pratique, il faudrait évidemment lire les données dans une base de données).

ServeurSocket.java

```
/*
 * ServeurSocket.java
 */
package clientserveursocket;

/**
 * @author Vilvens
 */
import java.io.*;
import java.net.*;
import java.util.*;
```

```

public class ServeurSocket
{
    private static Hashtable nbreActions = new Hashtable();
    static
    {
        getNbreActions().put("Vilvens", 1520); getNbreActions().put("Charlet", 5210);
        getNbreActions().put("Madani", 541); getNbreActions().put("Wagner", 1519);
        getNbreActions().put("Mercenier", 15230); getNbreActions().put("Kuty", 300);
    }
    public static Hashtable getNbreActions() { return nbreActions; }

    public static void main(String[] args)
    {
        int port = 50000;
        ServerSocket SSocket;
        Socket CSocket;

        SSocket = null; CSocket = null;
        try
        {
            SSocket = new ServerSocket(port);
        }
        catch (IOException e)
        {
            System.err.println("Erreur de port d'écoute ! ? [" + e + "]"); System.exit(1);
        }
        System.out.println("Serveur en attente");

        try
        {
            CSocket = SSocket.accept();
        }
        catch (SocketException e)
        {
            System.err.println("Accept interrompu ! ? [" + e + "]");
        }
        catch (IOException e)
        {
            System.err.println("Erreur d'accept ! ? [" + e + "]"); System.exit(1);
        }

        try
        {
            System.out.println("Serveur a reçu connexion");
            DataInputStream dis = new DataInputStream(CSocket.getInputStream());
            DataOutputStream dos = new DataOutputStream(CSocket.getOutputStream());
        }

        String inNom, outReponse;
        int inQuantite;
    }
}

```

```

while ( !(inNom = dis.readUTF()).equals("FIN"))
{
    try
    {
        inQuantite = dis.readInt();
    }
    catch (NumberFormatException e)
    {
        System.err.println("Erreur ! La quantité lue n'est pas un nombre [" + e + "]");
        continue; // Denys-like
    }
    System.out.println("Requête reçue = " + inNom + "(" + inQuantite + ")");
    int quantiteRestante = 0;
    if (getNbreActions().containsKey(inNom))
    {
        quantiteRestante = ((Integer) getNbreActions().get(inNom)) - inQuantite;
        dos.writeUTF(inNom);
        dos.writeInt(quantiteRestante);
        System.out.println("Serveur a envoyé réponse positive");
    }
    else
    {
        dos.writeUTF(inNom + "INCONNU");
        dos.writeInt(-1);
        System.out.println("Serveur a envoyé réponse négative");
    }
    dos.flush();
}
dis.close(); dos.close();
CSocket.close(); SSocket.close();
System.out.println("Communication terminée");
}
catch (EOFException e)
{
    System.err.println("Le client a terminé la communication");
}
catch (IOException e)
{
    System.err.println("Erreur ! ? [" + e + "]");
}
}
}
}

```

Ce qui donne :

Serveur en attente

Serveur en attente

Requête recue = Mercenier(25)

Requête reçue - Mercenier(25)
Serveur a envoyé réponse positive

Le client a terminé la communication

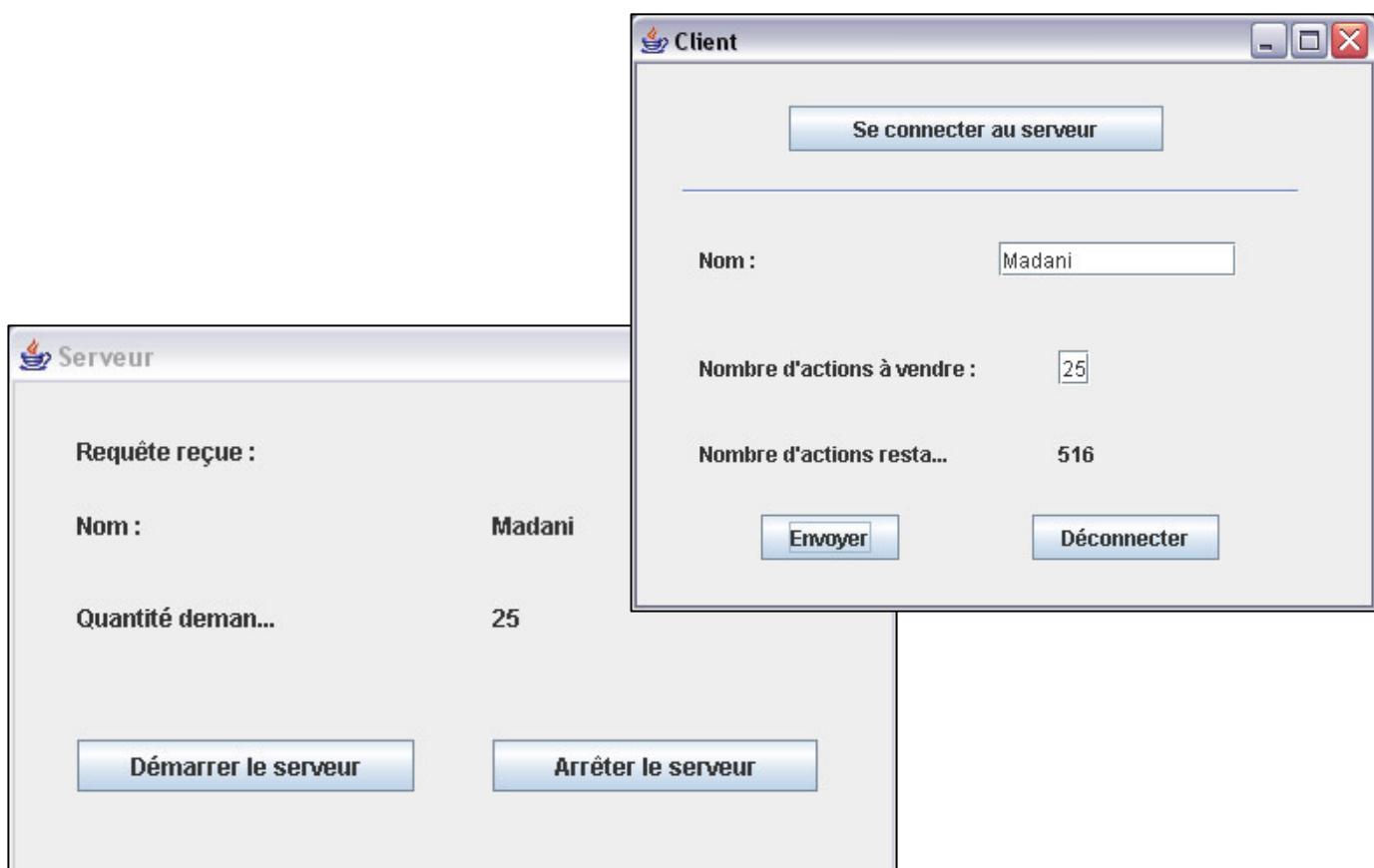
Remarques

1) Et si il y avait plusieurs clients potentiels ? Le serveur créerait un thread pour le client qui vient de se connecter. Le thread réaliserait le traitement tandis que le serveur se remettrait à l'écoute ... Nous allons en reparler plus loin.

2) On pourrait penser que la communication à partir d'une applet est du même style que celle à partir d'une application. Mais ce serait oublier les restrictions qui limitent les actions des applets. En fait, la seule communication permise à une applet est une connexion avec son serveur HTTP. Une application qui se trouve sur ce serveur et qui communique avec l'applet s'appelle alors une **servlet**. Elle n'est pas forcément écrite en Java (cela peut être en C ou C++), pour peu qu'elle manipule le protocole TCP/IP. Nous en reparlerons plus loin aussi ...

4. Une communication client-serveur basée TCP en mode GUI

Nous allons évidemment très vite être tentés par des clients et serveurs pourvus d'un GUI. Nous pensons donc à quelque chose de ce genre :



Cela ne change pas fondamentalement la programmation réseau. Jetons cependant un coup d'œil ...

4.1 Le client GUI

On peut se rendre compte que, par rapport à l'application client en mode console, la seule différence est d'avoir déplacé les éléments de code dans les méthodes de gestion d'appui sur les boutons "Connecter", "Envoyer" et "Déconnecter" :

FenClientSocket.java

```
/*
 * FenClientSocket.java
 */

package clientserveursocket;

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

/**
 * @author Vilvens
 */

public class FenClientSocket extends javax.swing.JFrame
{
    private Socket cliSock;
    private DataInputStream dis = null;
    private DataOutputStream dos = null;

    public FenClientSocket()
    {
        initComponents();
        cliSock=null; dis =null; dos = null;
    }

    private void initComponents() { ... }

    private void BDeconnecterActionPerformed(java.awt.event.ActionEvent evt)
    {
        try
        {
            dos.writeUTF("FIN");
            dos.writeInt(-2);
            cliSock.close();
        }
        catch (IOException e)
        {
            System.err.println("Erreur ! Pas de connexion ? [" + e + "]");
        }
        System.exit(0);
    }
}
```

```

private void BEnvoyerActionPerformed(java.awt.event.ActionEvent evt)
{
    String outNom = TFNom.getText();
    int outQuantite = Integer.parseInt(TFNOMBRE.getText());
    String reponse = null;
    int inQuantiteRestante = 0;
    System.out.println("outNom = " + outNom + " et outQuantite =" + outQuantite);
    if (!outNom.equals("") && outQuantite>0 && dos!=null && dis!=null)
    {
        System.out.println("Client au travail");
        try
        {
            dos.writeUTF(outNom);
            dos.writeInt(outQuantite);
            reponse = dis.readUTF();
            inQuantiteRestante = dis.readInt();
            TFNOM.setText(reponse);
            LQuantiteRestante.setText(String.valueOf(inQuantiteRestante));
        }
        catch (IOException e)
        {
            System.err.println("Erreur ! Pas de connexion ? [" + e + "]");
        }
    }
}

private void BConnecterActionPerformed(java.awt.event.ActionEvent evt)
{
    try
    {
        cliSock = new Socket("192.168.1.8", 50000);
        System.out.println("Client connecté : " + cliSock.getInetAddress().toString());
    }
    catch (UnknownHostException e)
    {
        System.err.println("Erreur ! Host non trouvé [" + e + "]");
    }
    catch (IOException e)
    {
        System.err.println("Erreur ! Pas de connexion ? [" + e + "]");
    }
    try
    {
        dis = new DataInputStream(cliSock.getInputStream());
        dos = new DataOutputStream(cliSock.getOutputStream());
        System.out.println("Flux créés");
    }
    catch (IOException e)
    {
        System.err.println("Erreur ! Pas de connexion ? [" + e + "]");
    }
}

public static void main(String args[])
{
}

```

```

java.awt.EventQueue.invokeLater(new Runnable()
{
    public void run()
    {
        new FenClientSocket().setVisible(true);
    }
});
}

private javax.swing.JButton BConnecter;
private javax.swing.JButton BDeconnecter;
private javax.swing.JButton BEnvoyer;
private javax.swing.JLabel LNom;
private javax.swing.JLabel LQuantiteRestante;
private javax.swing.JTextField TFNom;
private javax.swing.JTextField TFNombre;
private javax.swing.JLabel jLabel1;
private javax.swing.JLabel jLabel2;
private javax.swing.JSeparator jSeparator1;
}
}

```

4.2 Le serveur GUI et son thread

On pourrait penser que la transformation graphique du serveur est similaire à celle du client. Cependant, une complication apparaît. En effet, **nous devons impérativement "threader" notre composante serveur**, car le côté bloquant de la méthode accept() a pour effet de "geler" le GUI si nous le programmons gentiment dans le constructeur de la fenêtre ...

Dans un sens, cela simplifie les choses puisqu'en gros, nous n'avons plus qu'à migrer notre code serveur de l'application console dans la méthode run() du thread. Mais il faudra tout de même tenir compte de l'interface graphique, représenté par une variable membre instance de la JFrame utilisée (classe FenServeurSocket). On se souviendra aussi de la discussion concernant l'**EventQueue** de la JVM (voir chapitre consacré aux threads).

ThreadServeur.java

```

/*
 * ThreadServeur.java
 */

package clientserveursocket;

/**
 *
 * @author Vilvens
 */

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

```

```
public class ThreadServeur extends Thread
{
    private int port;
    private ServerSocket SSocket;
    private Socket CSocket;

    private FenServeurSocket fenetreApplication;

    public ThreadServeur(int p, FenServeurSocket fss)
    {
        port = p; fenetreApplication = fss;
    }

    public void run()
    {
        SSocket = null; CSocket = null;
        try
        {
            SSocket = new ServerSocket(port);
        }
        catch (IOException e)
        {
            System.err.println("Erreur de port d'écoute ! ? [" + e + "]");
            System.exit(1);
        }
        System.out.println("Serveur en attente");
        try
        {
            CSocket = SSocket.accept();
        }
        catch (SocketException e)
        {
            System.err.println("Accept interrompu ! ? [" + e + "]");
        }
        catch (IOException e)
        {
            System.err.println("Erreur d'accept ! ? [" + e + "]");
            System.exit(1);
        }

        try
        {
            System.out.println("Serveur a reçu connexion");
            DataInputStream dis = new DataInputStream(CSocket.getInputStream());
            DataOutputStream dos = new DataOutputStream(CSocket.getOutputStream());

            String inNom, outReponse;
            int inQuantite;
```

```

while ( !(inNom = dis.readUTF()).equals("FIN"))
{
    try
    {
        inQuantite = dis.readInt();
    }
    catch (NumberFormatException e)
    {
        System.err.println("Erreur ! La quantité lue n'est pas un nombre [" + e + "]");
        continue; // Denys-like
    }
    System.out.println("Requête reçue = " + inNom + "(" + inQuantite + ")");
    fenetreApplication.getLNom().setText(inNom);
    fenetreApplication.getLQuantiteDemandee().setText(
        String.valueOf(inQuantite));
    int quantiteRestante = 0;
    if (FenServeurSocket.getNbreActions().containsKey(inNom))
    {
        quantiteRestante = ((Integer)FenServeurSocket.getNbreActions().
            get(inNom)) - inQuantite;
        dos.writeUTF(inNom);
        dos.writeInt(quantiteRestante);
        System.out.println("Serveur a envoyé réponse positive");
    }
    else
    {
        dos.writeUTF(inNom + "INCONNU");
        dos.writeInt(-1);
        System.out.println("Serveur a envoyé réponse négative");
    }
    dos.flush();
}
dis.close(); dos.close();

CSocket.close();
System.out.println("Client déconnecté");
}
catch (IOException e)
{
    System.err.println("Erreur ! ? [" + e + "]");
}
}

public void doStop()
{
    try
    {
        SSocket.close();
    }
    catch (IOException e)
}

```

```

    {
        System.err.println("Erreur ! ? [" + e + "]");
    }
    System.out.println("Serveur déconnecté");
    stop();
}
}

```

Ceci devient d'autant plus clair que l'on connaît le code du GUI proprement dit :

FenServeurSocket.java

```

/*
 * FenServeurSocket.java
 */

package clientserveursocket;

/**
 * @author Vilvens
 */

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

public class FenServeurSocket extends javax.swing.JFrame
{
    private static Hashtable nbreActions = new Hashtable();
    static
    {
        getNbreActions().put("Vilvens", 1520); getNbreActions().put("Charlet", 5210);
        getNbreActions().put("Madani", 541); getNbreActions().put("Wagner", 1519);
        getNbreActions().put("Mercenier", 15230); getNbreActions().put("Kuty", 300);
    }
    public static Hashtable getNbreActions()
    {
        return nbreActions;
    }

    private int port;
    private ThreadServeur ts;

    public FenServeurSocket(int p)
    {
        initComponents();
        port = p;
    }

    private void initComponents() { ... }
}

```

```

private void BArreterActionPerformed(java.awt.event.ActionEvent evt)
{
    ts.doStop();
    System.exit(0);
}

private void BDemarrerActionPerformed(java.awt.event.ActionEvent evt)
{
    ts = new ThreadServeur(port, this);
    ts.start();
}

public static void main(String args[])
{
    java.awt.EventQueue.invokeLater(new Runnable()
    {
        public void run()
        {
            new FenServeurSocket(50000).setVisible(true);
        }
    });
}

public javax.swing.JLabel getLNom() { return LNOM; }
public javax.swing.JLabel getLQuantiteDemandee() { return LQuantiteDemandee; }

private javax.swing.JButton BArreter;
private javax.swing.JButton BDemarrer;
private javax.swing.JLabel LAnnonce;
private javax.swing.JLabel JLabel LNOM;
private javax.swing.JLabel LNOMA;
private javax.swing.JLabel LQuantiteA;
private javax.swing.JLabel JLabel LQuantiteDemandee;
private javax.swing.JButton jButton1;
}

```

Le thread contient manifestement l'essentiel de la logique tandis que le fenêtre de l'application se charge de la visualisation – cela fait penser au modèle document-view ou model-view-controler version condensée (le model et le controler sont confondus).

5. Des caractères ou des bytes sur le réseau

5.1 L'échange de caractères

Lorsque les échanges réseaux se limitent à des messages sous forme de chaînes de caractères constituant des "lignes" au sens habituel du terme, il est "deprecated" d'utiliser des flux binaires alors que l'on manipule clairement du texte : les flux caractères sont donc de mise et la notion de ligne justifie alors d'utiliser des **BufferedReader** et **BufferedWriter**. Dans le programme suivant, un client se connecte sur un serveur situé sur la machine nommée "myriam" ; le serveur est supposé écouter sur le port 6000. Le serveur envoie un nom et le

client propose un prénom sensé correspondre. Le serveur vérifie et envoie la chaîne "FIN." pour terminer la connexion. Des tels client et serveur peuvent être écrits comme suit.

ServerSocket.java (version orientée caractères)

```

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

public class ServerSocket
{
    public static void main(String[] args)
    {
        ServerSocket SSocket = null;
        try
        {
            SSocket = new ServerSocket(6000);
        }
        catch (IOException e)
        { System.err.println("Erreur de port d'écoute ! ? [" + e + "]"); System.exit(1);}

        Socket CSocket = null;
        System.out.println("Serveur en attente");
        try
        {
            CSocket = SSocket.accept();
        }
        catch (IOException e)
        { System.err.println("Erreur d'accept ! ? [" + e + "]"); System.exit(1);}

        try
        {
            BufferedReader dis =
                new BufferedReader (new InputStreamReader (CSocket.getInputStream()));
            BufferedWriter dos =
                new BufferedWriter (new OutputStreamWriter (CSocket.getOutputStream()));

            String inLigne, outLigne;

            String noms[] = {"vil", "char", "cler", "del", "FIN."};
            String prenoms[] = { "claude", "christophe", "carine", "pierre", "FIN."};
            int pos=0;
            String rep = null;

            outLigne = noms[pos];
            dos.write(outLigne + "\n");dos.flush();

            while (noms[pos++] != "FIN." && (inLigne=dis.readLine()) != null)
            {
                System.out.println("Réponse reçue = " + inLigne + "(" + inLigne.length() + ")");
                if (inLigne.equals(prenoms[pos-1])) rep="OK "; else rep="FAUX ";
        }
    }
}

```

```

        System.out.println("Serveur en " + pos);
        outLigne = rep + " -- suite : (" + pos + ")=" + noms[pos];
        dos.write(outLigne + "\n");dos.flush();
    }
    dis.close();dos.close();CSocket.close();SSocket.close();
    System.out.println("Serveur déconnecté");
}
catch (IOException e)
{
    System.err.println("Erreur ! ? [" + e + "]");
}
}
}

```

Nous utilisons donc ici nos flux dans **un modèle à trois couches** : bytes issus d'une socket, transformations en caractères selon le charset et concentration des caractères en ligne. Le client pratique de même :

ClientSocket (version orientée caractères)

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

public class ClientSocket
{
    public static void main(String[] args)
    {
        Socket cliSock = null;
        BufferedReader dis=null; BufferedWriter dos=null;
        String ligneDuServeur;

        try
        {
            cliSock = new Socket("myriam", 6000);
            System.out.println(cliSock.getInetAddress().toString());

            dis = new BufferedReader (new InputStreamReader(cliSock.getInputStream()));
            dos = new BufferedWriter(new OutputStreamWriter(cliSock.getOutputStream()));
        }
        catch (UnknownHostException e)
        { System.err.println("Erreur ! Host non trouvé [" + e + "]"); }
        catch (IOException e)
        { System.err.println("Erreur ! Pas de connexion ? [" + e + "]"); }

        if (cliSock==null || dis==null || dos==null) System.exit(1);

        try
        {
            StringBuffer buf = new StringBuffer(50);
            int c;
```

```

while ( (ligneDuServeur=dis.readLine()) != null )
{
    System.out.println("Reçu du serveur : " + ligneDuServeur);
    if (ligneDuServeur.indexOf("FIN.")===-1)
    {
        while((c = System.in.read())!="\n") if (c != '\r') buf.append((char)c);
        System.out.println("Envoyé au serveur : " + buf.toString());
        dos.write(buf.toString()+"\n");
        dos.flush();
        buf.setLength(0);
    }
}
dos.close(); dis.close();
cliSock.close();
System.out.println("Client déconnecté");
}
catch (UnknownHostException e)
{
    System.err.println("Erreur ! Host non trouvé [" + e + "]");
}
catch (IOException e)
{
    System.err.println("Erreur ! Pas de connexion ? [" + e + "]");
}
}
}

```

5.2 L'échange de bytes

A priori, les applications ci-dessus remplissent parfaitement leur rôle. Cependant, ceci n'est vrai que *parce que les deux extrémités de la connexion sont programmés en Java* : on use ainsi des mêmes conventions de codage, notamment Unicode ou UTF.

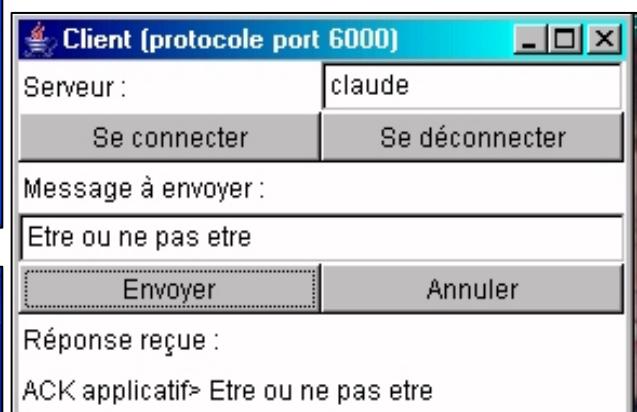
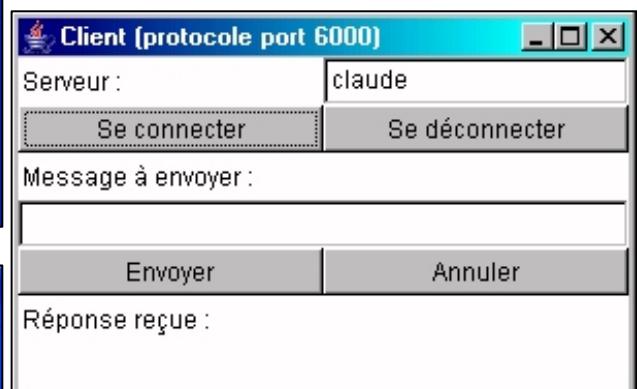
Mais qu'en serait-il si le client (ou le serveur) était, par exemple, écrit en C ? Ce langage ne connaît que les bytes (qui, pour lui, sont synonymes de caractères), si bien qu'il faut que les deux extrémités de la connexion s'expriment en bytes.

Considérons donc un client et un serveur qui échangent du texte représenté par des bytes et non plus par des objets String. Disons que leur interface et une séquence d'exécution seront les suivants :

serveur (machine claude – Windows XP)



client (machine myriam – Windows 98)



Réponse reçue :
ACK applicatif> Etre ou ne pas etre

Le message "END_OF_CONNEXION" reçu par le serveur correspond à l'action "Se déconnecter" déclenchée par le client.

Comme annoncé, le serveur et le client envoient et reçoivent cette fois des bytes, en utilisant les méthodes :

- ◆ public final byte **readByte()** throws IOException
de la classe DataInputStream
- ◆ public byte[] **getBytes()**,
de la classe String

Le serveur, dans une version simple, peut s'écrire :

FenServer.java

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

public class FenServer extends java.awt.Frame
{
    ServerSocket SSocket;
    Socket CSocket;

    public FenServer()
    {
        initComponents();
    }

    private void initComponents()
    {
        ZTServeur = new java.awt.Label(); panel1 = new java.awt.Panel();
        BEnMarche = new java.awt.Button(); BArret = new java.awt.Button();
        label2 = new java.awt.Label(); ZTClient = new java.awt.Label();
        label1 = new java.awt.Label(); ZTMessage = new java.awt.Label();
        setLayout(new java.awt.GridLayout(6, 1));
        setTitle("Serveur (protocole port 6000)");
        addWindowListener(new java.awt.event.WindowAdapter() {
            public void windowClosing(java.awt.event.WindowEvent evt) {
                exitForm(evt);
            }
        });
    }

    ZTServeur.setName("ZTServeur");
    ZTServeur.setText("Serveur \u00e0 l'arr\u00eat"); add(ZTServeur);

    panel1.setLayout(new java.awt.GridLayout(1, 2));
    BEnMarche.setLabel("D\u00e9marrage");
    BEnMarche.addActionListener(new java.awt.event.ActionListener() {
        public void actionPerformed(java.awt.event.ActionEvent evt) {
            BEnMarcheActionPerformed(evt);
        }
    });
    panel1.add(BEnMarche);
    BArret.setLabel("Arr\u00eat");
    BArret.addActionListener(new java.awt.event.ActionListener() {
        public void actionPerformed(java.awt.event.ActionEvent evt) {
            BArretActionPerformed(evt);
        }
    });
    panel1.add(BArret); add(panel1);

    label2.setText("Client connect\u00e9:");
    add(label2);
```

```

ZTClient.setName("ZTClient"); ZTClient.setText("?");
label1.setText("Message re\u00e9u :"); add(label1); add(ZTMessage);
pack();
}

private void BArretActionPerformed(java.awt.event.ActionEvent evt)
{
    try
    {
        CSocket.close();SSocket.close();
    }
    catch (IOException e)
    {
        System.err.println("Erreur ! ? [" + e + "]");
    }

ZTServeur.setText("Serveur arrêté");
System.out.println("Serveur arrêté");
}

private void BEnMarcheActionPerformed(java.awt.event.ActionEvent evt)
{
    SSocket = null;
    try
    {
        SSocket = new ServerSocket(6000);
    }
    catch (IOException e)
    {
        System.err.println("Erreur de port d'écoute ! ? [" + e + "]");
        System.exit(1);
    }

    CSocket = null;
    DataInputStream dis = null; DataOutputStream dos = null;
    System.out.println("Serveur en attente"); ZTServeur.setText("Serveur en marche");
    try
    {
        CSocket = SSocket.accept();
        dis = new DataInputStream(new BufferedInputStream(CSocket.getInputStream()));
        dos = new DataOutputStream(new BufferedOutputStream(CSocket.getOutputStream()));

    }
    catch (IOException e)
    {
        System.err.println("Erreur d'accept ! ? [" + e + "]");
        System.exit(1);
    }

    System.out.println("Une connexion client acceptée");
}

```

```

InetSocketAddress ad = (InetSocketAddress)CSocket.getRemoteSocketAddress();
String idClient = ad.getHostName() + " (" + CSocket.getInetAddress().getHostAddress() +
")";
ZTClient.setText(idClient);
System.out.println("Connexion provenant de : " + idClient);

StringBuffer message = new StringBuffer();
boolean fini = false;
byte b=0; int cpt = 0;
do
{
    message.setLength(0);
    try
    {
        while ( (b=dis.readByte())!= (byte)'\n' )
        {
            if (b!='\n') message.append((char) b);
            System.out.println("byte reçu = "+b);
        }
    }
    catch (IOException e)
    {
        System.out.println("Erreur de lecture = " + e.getMessage());
    }

    System.out.println("dernier byte reçu = "+b); cpt++;
    System.out.println (cpt + ". Message reçu = " + message.toString().trim());
    ZTMessage.setTextt(message.toString().trim());
    fini = "END_OF_CONNEXION".equals(message.toString().trim()) || cpt>5;
    System.out.println("fini = " + fini);
    String reponse = "ACK applicatif> " + message + "\n";
    try
    {
        if (!fini) dos.write(reponse.getBytes()); dos.flush();
    }
    catch (IOException e)
    {
        System.out.println("Erreur de lecture = " + e.getMessage());
    }
}
while (!fini);
try
{
    dis.close();dos.close();
}
catch (IOException e)
{
    System.out.println("Erreur de fermeture = " + e.getMessage());
}
}

```

```
private void exitForm(java.awt.event.WindowEvent evt) { System.exit(0); }

public static void main(String args[])
{
    new FenServer().show();
}

private java.awt.Label ZTMessage;
private java.awt.Label ZTClient;
private java.awt.Panel panel1;
private java.awt.Label label2;
private java.awt.Button BEnMarche;
private java.awt.Label ZTServeur;
private java.awt.Button BArret;
private java.awt.Label label1;
}
```

Les affichages de bytes sur la console n'ont d'autre but que de remarquer que le caractère terminal des chaînes est celui de code ASCII 10 (le '\n' a été tronqué sur un byte) – la méthode

String **trim()**

qui fournit une chaîne nettoyée des blancs (c'est-à-dire de code <= 32) initiaux et terminaux ainsi que les caractères nuls \x00 se révèle bien utile ici.

Signalons encore qu'à partir du JDK 1.4³, comme on peut le voir dans le code du serveur, il est possible d'obtenir des informations sur le client qui s'est connecté au moyen des méthodes de la classe Socket :

- ◆ public InetAddress **getInetAddress()**
déjà rencontrée, où la classe InetAddress comporte notamment la méthode

public String **getHostAddress()**

- ◆ public SocketAddress **getRemoteSocketAddress()**

En fait, SocketAddress est une classe abstraite indépendante du protocole utilisé et c'est la classe dérivée **InetSocketAddress** qui est plus particulièrement adaptée à IP, avec par exemple la méthode

public final String **getHostName()**

Le client se décline évidemment plus simplement :

³ quand on connaît quelque peu la programmation TCP/IP sous UNIX, c'est à se demander pourquoi il a fallu attendre la version 1.4 pour disposer de ces renseignements ;-) ...

FenClient.java

```

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

public class FenClient extends java.awt.Frame
{
    public final static int PORT_ECOUTE_SERVEUR = 6000;
    public final static String NOM_SERVEUR = "claude";

    Socket cliSock = null; /* Initialisations indispensables */
    DataInputStream dis=null;
    DataOutputStream dos=null;
    StringBuffer message = null;
    boolean fini = false;

    public FenClient() { initComponents(); }

    private void initComponents()
    {
        panel1 = new java.awt.Panel();
        label1 = new java.awt.Label(); ZEServeur = new java.awt.TextField();
        panel2 = new java.awt.Panel();
        BConnecter = new java.awt.Button(); BDeconnecter = new java.awt.Button();
        label2 = new java.awt.Label(); ZEMessage = new java.awt.TextField();
        panel3 = new java.awt.Panel();
        BEnvoyer = new java.awt.Button(); BAnnuler = new java.awt.Button();
        label3 = new java.awt.Label(); ZTReponse = new java.awt.Label();

        setLayout(new java.awt.GridLayout(7, 1));

        setTitle("Client (protocole port 6000)");
        addWindowListener(new java.awt.event.WindowAdapter() {
            public void windowClosing(java.awt.event.WindowEvent evt) {
                exitForm(evt);
            }
        });

        panel1.setLayout(new java.awt.GridLayout(1, 2));
        label1.setText("Serveur :"); panel1.add(label1);
        panel1.add(ZEServeur);
        add(panel1);

        panel2.setLayout(new java.awt.GridLayout(1, 2));
        BConnecter.setLabel("Se connecter");
        BConnecter.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                BConnecterActionPerformed(evt);
            }
        });
    }
}

```

```

panel2.add(BConnecter);
BDeconnecter.setLabel("Se déconnecter");
BDeconnecter.addActionListener(new java.awt.event.ActionListener() {
    public void actionPerformed(java.awt.event.ActionEvent evt) {
        BDeconnecterActionPerformed(evt);
    }
});

panel2.add(BDeconnecter);
add(panel2);
label2.setText("Message à envoyer :");
add(label2);
add(ZEMessage);

panel3.setLayout(new java.awt.GridLayout(1, 2));
BEnvoyer.setLabel("Envoyer");
BEnvoyer.addActionListener(new java.awt.event.ActionListener() {
    public void actionPerformed(java.awt.event.ActionEvent evt) {
        BEnvoyerActionPerformed(evt);
    }
});
panel3.add(BEnvoyer);
BAnnuler.setLabel("Annuler"); panel3.add(BAnnuler);
add(panel3);
label3.setText("Réponse reçue :");
add(label3);
add(ZTReponse);

pack();
}

private void BDeconnecterActionPerformed(java.awt.event.ActionEvent evt)
{
    try
    {
        BEnvoyer.setEnabled(false);
        message = new StringBuffer("END_OF_CONNEXION\n");

        dos.write(message.toString().getBytes());
        dos.flush();
        dos.close(); dis.close(); cliSock.close();
        System.out.println("Client déconnecté");
    }
    catch (IOException e)
    {
        System.err.println("Erreur de fermeture ? [" + e + "]");
    }
}

```

```

private void BEnvoyerActionPerformed(java.awt.event.ActionEvent evt)
{
    try
    {
        int cpt=0, c;
        byte b;

        cpt++;
        System.out.println("Message numero " + cpt + " à envoyer au serveur : ");
        message = new StringBuffer(ZEMessage.getText() + "\n");

        dos.write(message.toString().getBytes()); dos.flush();
        message.setLength(0);
        System.out.println("Attente de la réponse du serveur");
        StringBuffer reponse = new StringBuffer();
        int cptCharServeur = 0;
        while((b = dis.readByte())!=((byte)\n))
        {
            System.out.println("byte lu : " + b); reponse.append((char)b);
        }
        System.out.println("Reçu du serveur : " + reponse.toString().trim());
        ZTReponse.setText(reponse.toString().trim());
    }
    catch (IOException e)
    {
        System.err.println("Erreur de communication ? [" + e + "]");
    }
}

private void BConnecterActionPerformed(java.awt.event.ActionEvent evt)
{
    try
    {
        String nomServeur = ZEServeur.getText();
        System.out.println("Tentative de connexion sur le serveur " + nomServeur);
        if (nomServeur.equals("")) nomServeur= NOM_SERVEUR;
        cliSock = new Socket(nomServeur, PORT_ECOUTE_SERVEUR);
        System.out.println(cliSock.getInetAddress().toString());
        dis = new DataInputStream(cliSock.getInputStream());
        dos = new DataOutputStream(cliSock.getOutputStream());
    }
    catch (UnknownHostException e)
    {
        System.err.println("Erreur ! Host non trouvé [" + e + "]");
    }
    catch (IOException e)
    {
        System.err.println("Erreur ! Pas de connexion ? [" + e + "]");
    }
}

```

```

private void exitForm(java.awt.event.WindowEvent evt) { System.exit(0); }

public static void main(String args[])
{
    new FenClient().show();
}

private java.awt.Button BDeconnecter;
private java.awt.Label label3;
private java.awt.Panel panel2;
private java.awt.Button BConnecter;
private java.awt.TextField ZEServeur;
private java.awt.Button BEnvoyer;
private java.awt.Panel panel1;
private java.awt.Label label2;
private java.awt.Button BAnnuler;
private java.awt.Panel panel3;
private java.awt.TextField ZEMessage;
private java.awt.Label ZTReponse;
private java.awt.Label label1;
}

```

Le terrain semble donc prêt pour le bas niveau. Passons à présent au point de vue opposé ...

6. La communication par sérialisation

6.1 Une information sérialisée

On se souviendra sans doute du procédé de sérialisation évoqué dans le chapitre des flux⁴, donnant aux objets la persistance par simple implémentation par leur classe de l'interface **Serializable**. Quand on y réfléchit, un **ObjectOutputStream** ou **ObjectInputStream** se construit sur un flux quelconque : et ***pourquoi pas sur un flux réseau ?*** Effectivement, cela fonctionne tout aussi bien !

Considérons donc une classe matérialisant l'information que nous souhaitons voir échangée sur le réseau entre un serveur et ses clients :

information.java

```

import java.io.*;

class information implements Serializable
{
    private int numero;
    private String texteInfo;
    private String origine;
    private String niveauImportance; // Prioritaire, Urgent, Normal, PourInfo
}

```

⁴ voir "Langage Java (I) : Programmation de base"

```

static public information FinDeCommunication =
    new information(-1,"fin de communication", "Serveur Claude", "Prioritaire");

public information(int n, String i, String o, String ni)
{
    numero=n; texteInfo=i; origine=o; niveauImportance=ni;
}

public boolean equals(information i)
{
    if (numero==i.numero && texteInfo.equals(i.texteInfo) && origine.equals(i.origine))
        return true;
    else return false;
}

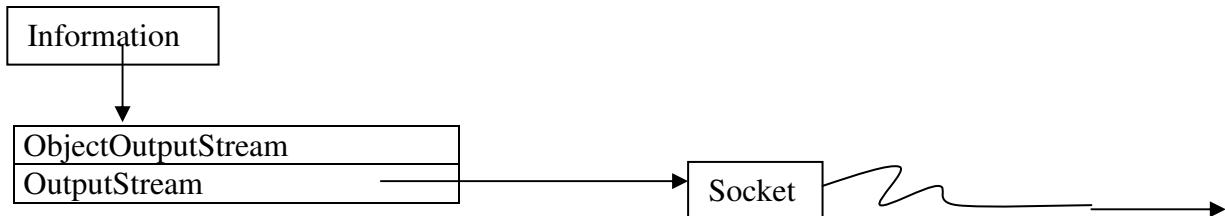
void affiche()
{
    System.out.println("[ " + origine + " ] " + numero + ". " + texteInfo + " (" +
        + niveauImportance + ")");
}
}

```

Rien de bien surprenant dans cette classe – remarquons cependant l'information particulière *FinDeCommunication*.

6.2 Le serveur d'objets sérialisés

Nous allons imaginer un serveur envoyant un nombre fixe (disons 5 par exemple) d'objets *information*, aux caractéristiques fixées aléatoirement. Ces objets sont écrits sur un *ObjectOutputStream* qui est construit sur l'*OutputStream* obtenu de la socket de service :



NetworkObjectServer.java

```

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

public class NetworkObjectServer
{
    public static void main(String args[])
    {
        // Mise en attente du serveur
        ServerSocket SSocket = null;
    }
}

```

```
try
{
    SSocket = new ServerSocket(50000);
}
catch (IOException e)
{
    System.err.println("Erreur de port d'écoute ! ? [" + e + "]");
    System.exit(1);
}

Socket CSocket = null;
System.out.println("Serveur en attente");

try
{
    CSocket = SSocket.accept();
}
catch (IOException e)
{
    System.err.println("Erreur d'accept ! ? [" + e.getMessage() + "]");
    System.exit(1);
}

// Prise en charge d'une connexion
ObjectOutputStream oos;
ObjectInputStream ois;
information i;

try
{
    oos = new ObjectOutputStream(CSocket.getOutputStream());
    ois = new ObjectInputStream(CSocket.getInputStream());
    for (int cpt=0; cpt<5; cpt++)
    {
        int r = new Double(Math.random()*10).intValue();
        String niveau;
        if (r%7==0) niveau = new String("Prioritaire");
        else if (r%3==0) niveau = new String("Urgent");
        else if (r%2==0) niveau = new String("Normal");
        else niveau = new String("PourInfo");

        i = new information(r, "essai", "Serveur Claude", niveau);
        oos.writeObject(i); oos.flush();
        System.out.println(" *** écriture d'une information");
        i.affiche();
        int temps = new Double(Math.random()*10).intValue()*1000;
        try { Thread.sleep(temps); } catch (InterruptedException e) { }
    }
}
```

```
    oos.writeObject(information.FinDeCommunication);
    oos.flush();
    oos.close();
}
catch (IOException e)
{
    System.err.println("Erreur ! ? [" + e.getMessage() + "]");
}
}
```

6.3 Le client lecteur d'objets sérialisés

On s'en doute, le client qui prétend recevoir les informations sérialisées réalisera les opérations symétriques à celles du serveur, mais en lecture :

NetworkObjectClient.java

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

public class NetworkObjectClient
{
    public static void main(String args[])
    {

        Socket cliSock = null;
        ObjectInputStream ois=null;
        ObjectOutputStream oos=null;

        // Connexion au serveur
        try
        {
            cliSock = new Socket("Claude", 50000);
            System.out.println(cliSock.getInetAddress().toString());

            ois = new ObjectInputStream(cliSock.getInputStream());
            oos = new ObjectOutputStream(cliSock.getOutputStream());
        }
        catch (UnknownHostException e)
        {
            System.err.println("Erreur ! Host non trouvé [" + e + "]");
        }
        catch (IOException e)
        {
            System.err.println("Erreur ! Pas de connexion ? [" + e + "]");
        }
        if (cliSock==null || ois==null || oos==null) System.exit(1);
    }
}
```

```
// Lecture des informations envoyées
try
{
    boolean fini = false;
    information i;
    do
    {
        i = (information)ois.readObject();
        System.out.println(" *** lecture faite");
        i.affiche();
    }
    while (!i.equals(information.FinDeCommunication));

    ois.close();
}
catch (ClassNotFoundException e)
{
    System.out.println("--- erreur sur la classe = " + e.getMessage());
}
catch (IOException e)
{
    System.out.println("--- erreur = " + e.getMessage());
}
}
```

Un exemple de communication sera, côté serveur :

```
Serveur en attente
*** écriture d'une information
[Serveur Claude] 8. essai (Normal)
*** écriture d'une information
[Serveur Claude] 0. essai (Prioritaire)
*** écriture d'une information
[Serveur Claude] 4. essai (Normal)
*** écriture d'une information
[Serveur Claude] 6. essai (Urgent)
*** écriture d'une information
[Serveur Claude] 3. essai (Urgent)
```

et du côté client :

```
Claude/192.168.2.2
*** lecture faite
[Serveur Claude] 8. essai (Normal)
*** lecture faite
[Serveur Claude] 0. essai (Prioritaire)
*** lecture faite
[Serveur Claude] 4. essai (Normal)
*** lecture faite
```

[Serveur Claude] 6. essai (Urgent)

*** lecture faite

[Serveur Claude] 3. essai (Urgent)

*** lecture faite

[Serveur Claude] -1. fin de communication (Prioritaire)

On peut vérifier, au moyen d'un logiciel d'analyse de trafic réseau ("sniffeur"), que ce qui passe sur le réseau est la réplique exacte de ce que l'on trouvait dans les fichiers de sérialisation.

Remarque

On l'aura compris, cette manière de programmer les communications réseaux permet l'encapsulation parfaite d'un protocole applicatif quelconque. On peut même aller plus loin ...

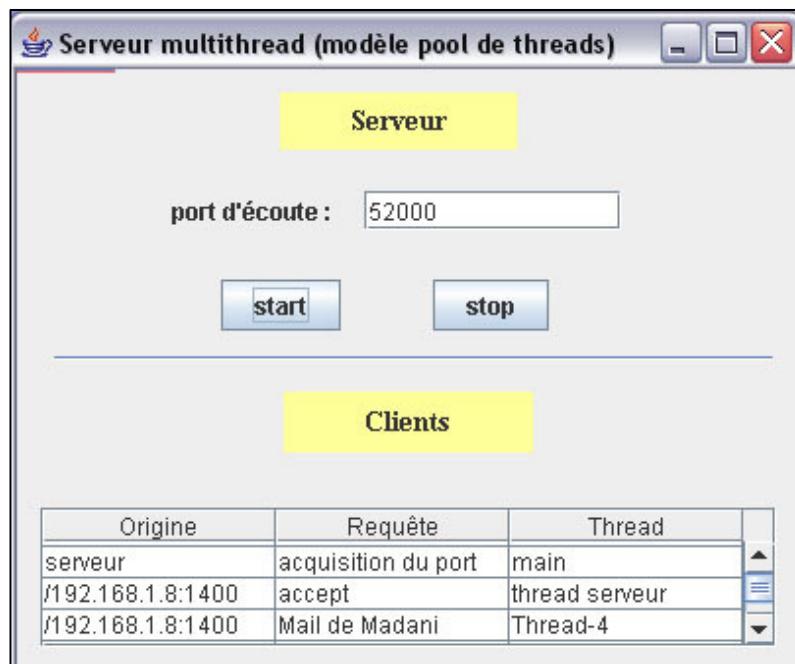
7. Un exemple simple de serveur multithread générique

7.1 Le contexte et les objectifs

Bien sûr, chaque contexte d'application dictera, après analyse, le type de modèle de threads et l'implémentation des protocoles pour la solution client-serveur à développer.

Cependant, nous allons donner ici un petit exemple de serveur multithread

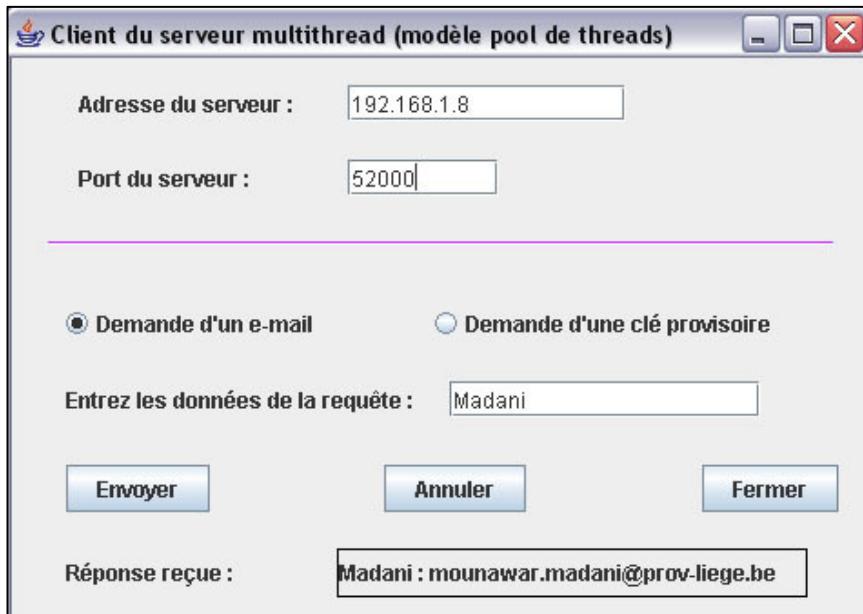
- ◆ basé sur le modèle du "pool de threads";
- ◆ à clients effectuant des requêtes de type "connexion-requête-réponse-déconnexion";
- ◆ essayant de prendre ses distances vis-à-vis du protocole effectivement utilisé (donc, ayant des velléités de générnicité);
- ◆ développé sous Netbeans 6.*;
- ◆ ayant le look suivant :



Le contexte sera fort simple. On considère qu'un client s'adresse à notre serveur :

- ◆ soit demande l'adresse e-mail d'une personne dont il fournit le nom;
- ◆ soit demande la génération d'une clé provisoire sur base de son mot de passe.

Il aura une apparence du type suivant :



Le protocole correspondant (appelons-le "**SUM**" : Simple Users Management) se déclinera donc selon ces deux types de requêtes.

7.2 Un thread serveur et des threads clients

Nous allons considérer que l'application serveur lance un thread chargé de gérer les connexions réseaux; le GUI de l'application ne sera donc pas inhibé par les actions réseaux et vice-versa. Ce thread

- ◆ se mettra donc à l'écoute sur un port donné;
- ◆ lancera au préalable les threads chargés de gérer les requêtes d'un client (on les devine en attente au départ, sans doute au moyen d'un `wait()`);
- ◆ à chaque connexion, lira la requête reçue sous forme d'un objet sérialisé;
- ◆ chargerà un thread du pool du traitement de la requête ...

à moins que tous les threads ne soient occupés ! D'où l'idée de placer les requêtes dans une file (par exemple – disons un réceptacle quelconque) que les threads consulteront en cas d'inactivité. Mais si nous mémorisons dans la file les requêtes elles-mêmes, nécessairement attachées à un protocole, nous perdons toute généricité. Pour sauvegarder cette dernière, nous allons plutôt imaginer que la file contient ce que le thread client doit faire, sans autre précision : autrement dit, la file va contenir des objets Runnable, dont la méthode `run()` sera celle que le thread exécutera. Celui-ci "threadera" donc un traitement dont il ne connaît pas la sémantique ☺ !

7.3 Quelques interfaces

Suite à cette réflexion, et toujours pour rester générique, nous allons définir le comportement de notre serveur (le thread serveur et les threads clients) en termes non pas d'instances de classes bien précises, mais d'objets implémentant des interfaces. Ainsi, notre

file (qui pourrait très bien être tout à fait autre chose) sera pensée de manière abstraite comme implémentant l'interface **SourceTaches** :

SourceTaches.java

```
/*
 * SourceTaches.java
 */

package serveurpoolthreads;

/**
 * @author Vilvens
 */

public interface SourceTaches
// synchronized ne s'utilise pas dans un interface
{
    public Runnable getTache() throws InterruptedException;
    public boolean existTaches();
    public void recordTache (Runnable r);
}
```

Un tel objet contiendra les tâches à exécuter sous forme d'objets Runnable. Il les mémorisera au moyen d'une méthode **recordTache()** et en fournira un (si il en existe une – sinon, on attendra : wait() ?) au moyen de sa méthode **getTache()**.

Comment associer une requête reçue à une telle tâche ? En considérant que les requêtes acceptées par notre serveur implémentent l'interface **Requete** :

Requete.java

```
/*
 * Requete.java
 */

package requetepoolthreads;

import java.net.*;

/**
 * @author Vilvens
 */

public interface Requete
{
    // Ce qui va être exécuté doit connaître la socket du client distant
    // ainsi que le GUI qui affiche les traces
    public Runnable createRunnable (Socket s, ConsoleServeur cs);
}
```

Logique : une requête doit être capable de générer l'objet Runnable qui la traite au moyen d'une méthode *createRunnable()*. On remarquera les deux paramètres :

- ◆ une socket (pour que l'on puisse toujours savoir d'où la requête provient);
- ◆ un objet désignant le GUI qui tracera les actions effectuées (si, du moins on le souhaite).

Ce GUI sera, entre autres, une implémentation de l'interface **ConsoleServeur** :

ConsoleServeur.java

```
/*
 * ConsoleServeur.java
 */

package serveurpoolthreads;

/**
 * @author Vilvens
 */
public interface ConsoleServeur
{
    public void TraceEvenements(String commentaire);
}
```

A priori, la méthode de traçage recevra une chaîne de caractères, que le serveur découpera à sa guise (par exemple, le symbole "#" séparera les différentes composantes à placer, par exemple, dans un tableau).

Pour terminer, il semble logique de définir aussi un interface **Reponse** :

Reponse.java

```
/*
 * Reponse.java
 */

package requetepoolthreads;

/**
 * @author Vilvens
 */

public interface Reponse
{
    public int getCode();
}
```

7.4 Le thread serveur générique

Nous pouvons à présent décrire le fonctionnement de notre thread serveur en termes de ces interfaces et aussi d'une classe ThreadClient dont il nous suffit de savoir pour l'instant que son constructeur réclame la source des tâches (logique puisque le thread va y trouver ce qu'il doit faire quand il est libre) ainsi qu'un nom :

ThreadServeur.java

```
/*
 * ThreadServeur.java
 */
package serveurpoolthreads;

import java.net.*;
import java.io.*;
import requetepoolthreads.Requete;

/**
 * @author Vilvens
 */
public class ThreadServeur extends Thread
{
    private int port;
    private SourceTaches tachesAExecuter;
    private ConsoleServeur guiApplication;
    private ServerSocket SSocket = null;

    public ThreadServeur(int p, SourceTaches st, ConsoleServeur fs)
    {
        port = p; tachesAExecuter = st; guiApplication = fs;
    }

    public void run()
    {
        try
        {
            SSocket = new ServerSocket(port);
        }
        catch (IOException e)
        {
            System.err.println("Erreur de port d'écoute ! ? [" + e + "]"); System.exit(1);
        }

        // Démarrage du pool de threads
        for (int i=0; i<3; i++) // 3 devrait être constante ou une propriété du fichier de config
        {
            ThreadClient thr = new ThreadClient (tachesAExecuter, "Thread du pool n°" +
                String.valueOf(i));
            thr.start();
        }
    }
}
```

```

// Mise en attente du serveur
Socket CSocket = null;

while (!isInterrupted())
{
    try
    {
        System.out.println("***** Serveur en attente");
        CSocket = SSocket.accept();
        guiApplication.TraceEvenements(CSocket.getRemoteSocketAddress().toString()+
            "#accept#thread serveur");
    }
    catch (IOException e)
    {
        System.err.println("Erreur d'accept ! ? [" + e.getMessage() + "]"); System.exit(1);
    }

ObjectInputStream ois=null;
Requete req = null;
try
{
    ois = new ObjectInputStream(CSocket.getInputStream());
    req = (Requete)ois.readObject();
    System.out.println("Requete lue par le serveur, instance de " +
        req.getClass().getName());
}
catch (ClassNotFoundException e)
{
    System.err.println("Erreur de def de classe [" + e.getMessage() + "]");
}
catch (IOException e)
{
    System.err.println("Erreur ? [" + e.getMessage() + "]");
}

Runnable travail = req.createRunnable(CSocket, guiApplication);
if (travail != null)
{
    tachesAExecuter.recordTache(travail);
    System.out.println("Travail mis dans la file");
}
else System.out.println("Pas de mise en file");
}
}

```

7.5 Le thread client générique

Comme prévu, un thread du pool boucle et tente de trouver du travail dans sa source de tâches :

ThreadClient.java

```
/*
 * ThreadClient.java
 */

package serveurpoolthreads;

/**
 * @author Vilvens
 */

public class ThreadClient extends Thread
{
    private SourceTaches tachesAExecuter;
    private String nom;

    private Runnable tacheEnCours;

    public ThreadClient(SourceTaches st, String n )
    {
        tachesAExecuter = st;
        nom = n;
    }

    public void runwhile (!isInterrupted())
        {
            try
            {
                System.out.println("Tread client avant get");
                tacheEnCours = tachesAExecuter.getTache();
            }
            catch (InterruptedException e)
            {
                System.out.println("Interruption : " + e.getMessage());
            }
            System.out.println("run de tachesencours");
            tacheEnCours.run();
        }
    }
}
```

Nous pouvons passer à présent au concret, c'est-à-dire aux implémentations.

7.6 L'implémentation de la source de tâches

Nous allons donc définir concrètement les idées générales exprimées par l'interface SourceTaches. Nous allons donc poser ici des choix, mais pas encore en fonction du protocole : simplement, nous allons déterminer comment les tâches seront mémorisées (ce sera au moyen d'une LinkedList) et comment les tâches vont être fournies. En fait :

- ◆ si aucune tâche n'est disponible, le thread qui invoquera la méthode getTache() doit rester en attente de l'apparition d'une telle tâche : un wait() fera l'affaire;
- ◆ quand une tâche est placée dans la liste (sans doute par le thread serveur), il faudra prévenir qu'une nouvelle tâche est disponible : un notify() fera l'affaire.

La classe ListeTaches reprend ces considérations :

ListeTaches.java

```
/*
 * ListeTaches.java
 */
package serveurpoolthreads;
/**
 * @author Vilvens
 */
import java.util.*;

public class ListeTaches implements SourceTaches
{
    private LinkedList listeTaches;
    public ListeTaches()
    {
        listeTaches = new LinkedList();
    }

    public synchronized Runnable getTache() throws InterruptedException
    {
        System.out.println("getTache avant wait");
        while (!existTaches()) wait();
        return (Runnable)listeTaches.remove();
    }

    public synchronized boolean existTaches()
    {
        return !listeTaches.isEmpty();
    }

    public synchronized void recordTache (Runnable r)
    {
        listeTaches.addLast(r);
        System.out.println("ListeTaches : tache dans la file");
        notify();
    }
}
```

On l'aura compris, cette classe est un moniteur ...

7.7 L'application serveur

Le serveur lui-même va donc proposer un GUI (et pour l'occasion implémenter l'interface ConsoleServeur – un JTable sera utilisé pour les affichages) et lancer le thread serveur en créant une liste de tâches et en se proposant comme ConsoleServeur pour ce thread :

FenAppServeur.java

```
/*
 * FenAppServeur.java
 *
 */

package serveurpoolthreads;

import java.util.*;
import javax.swing.*;
import javax.swing.table.*;
import java.net.*;
import java.io.*;

/**
 * @author Vilvens
 */

public class FenAppServeur extends javax.swing.JFrame implements ConsoleServeur
{
    private int port;
    public FenAppServeur()
    {
        initComponents();
        TraceEvenements("serveur#initialisation#main");
    }
    private void initComponents()
    { ... // festival de GroupLayouts ! }

    private void BStartActionPerfomed(java.awt.event.ActionEvent evt)
    {
        port = Integer.parseInt(TFPort.getText());
        TraceEvenements("serveur#acquisition du port#main");
        ThreadServeur ts = new ThreadServeur(port, new ListeTaches(), this);
        ts.start();
    }

    public static void main(String args[])
    {
        java.awt.EventQueue.invokeLater(new Runnable() {
            public void run() {
```

```

        new FenAppServeur().setVisible(true);
    }
});
}

public void TraceEvenements(String commentaire)
{
    Vector ligne = new Vector();
    StringTokenizer parser = new StringTokenizer(commentaire, "#");
    while (parser.hasMoreTokens())
        ligne.add(parser.nextToken());
    DefaultTableModel dtm = (DefaultTableModel)TableauEvenements.getModel();
    dtm.insertRow(dtm.getRowCount(),ligne);
}

private javax.swing.JButton BStart;
private javax.swing.JButton BStop;
private javax.swing.JScrollPane ScrollPaneTableauEvenements;
private javax.swing.JTextField TFPort;
private javax.swing.JTable TableauEvenements;
...
}

```

Mais il n'a toujours pas été question du protocole SUM ! Nous allons donc maintenant, et seulement maintenant, y penser.

7.8 Les requêtes du protocole

Une requête de SUM va donc

- ◆ définir des constantes associées au protocole;
- ◆ définir l'accès aux données; ici, tout se trouve dans des containers statiques – en pratique, bien entendu, il faudra prévoir des accès à des bases de données;
- ◆ implémenter l'interface Requete, donc la méthode createRunnable() : celle-ci crée un objet Runnable à la volée, lequel appelle dans sa méthode run() une méthode dédiacée à un traitement (soit la recherche de l'e-mail, soit la génération d'une clé); bien sûr, ces méthodes qui, en définitive, implémentent le protocole SUM, envoient une réponse qui est une instance de la classe ReponseSUM implémentant Reponse;
- ◆ implémenter l'interface Serializable pour l'envoi sur le réseau..

RequeteSUM.java

```
/*
 * RequeteSUM.java
 */
```

```
package ProtocoleSUM;
```

```
import java.io.*;
import java.util.*;
import java.net.*;
```

```

import serveurpoolthreads.*;
import requetepoolthreads.Requete;

/**
 * @author Vilvens
 */
public class RequeteSUM implements Requete, Serializable
{
    public static int REQUEST_E_MAIL = 1;
    public static int REQUEST_TEMPORARY_KEY = 2;
    public static Hashtable tableMails = new Hashtable();
    static
    {
        tableMails.put("Vilvens", "claude.vilvens@prov-liege.be");
        tableMails.put("Charlet", "christophe.charlet@prov-liege.be");
        tableMails.put("Madani", "mounawar.madani@prov-liege.be");
        tableMails.put("Wagner", "jean-marc.wagner@prov-liege.be");
    }
    public static Hashtable tablePwdNoms = new Hashtable();
    static
    {
        tablePwdNoms.put("GrosZZ", "Vilvens");
        tablePwdNoms.put("GrosRouteur", "Charlet");
        tablePwdNoms.put("GrosseVoiture", "Madani");
        tablePwdNoms.put("GrosCerveau", "Wagner");
    }

    private int type;
    private String chargeUtile;
    private Socket socketClient;

    public RequeteSUM(int t, String chu)
    {
        type = t; setChargeUtile(chu);
    }
    public RequeteSUM(int t, String chu, Socket s)
    {
        type = t; setChargeUtile(chu); socketClient = s;
    }

    public Runnable createRunnable (final Socket s, final ConsoleServeur cs)
    {
        if (type==REQUEST_E_MAIL)
            return new Runnable()
        {
            public void run()
            {
                traiteRequeteEMail(s, cs);
            }
        };
    }
}

```

```

else if (type==REQUEST_TEMPORARY_KEY)
    return new Runnable()
    {
        public void run()
        {
            traiteRequeteKey(s, cs);
        }
    };
else return null;
}

private void traiteRequeteEmail(Socket sock, ConsoleServeur cs)
{
    // Affichage des informations
    String adresseDistante = sock.getRemoteSocketAddress().toString();
    System.out.println("Début de traiteRequete : adresse distante = " + adresseDistante);
    // la charge utile est le nom du client
    String eMail = (String)tableMails.get(getChargeUtile());
    cs.TraceEvenements(adresseDistante+"#Mail de "+
        getChargeUtile()+"#" + Thread.currentThread().getName());
    if (eMail != null)
        System.out.println("E-Mail trouvé pour " + getChargeUtile());
    else
    {
        System.out.println("E-Mail non trouvé pour " + getChargeUtile() + " : " + eMail);
        eMail="?@?";
    }
    // Construction d'une réponse
    ReponseSUM rep = new ReponseSUM(ReponseSUM.EMAIL_OK, getChargeUtile() +
    " : " + eMail);

    ObjectOutputStream oos;
    try
    {
        oos = new ObjectOutputStream(sock.getOutputStream());
        oos.writeObject(rep); oos.flush();
        oos.close();
    }
    catch (IOException e)
    {
        System.err.println("Erreur réseau ? [" + e.getMessage() + "]");
    }
}

private void traiteRequeteKey(Socket sock, ConsoleServeur cs)
{
    // TO DO ;-)
}

public String getChargeUtile() { return chargeUtile; }

```

```
public void setChargeUtile(String chargeUtile)
{
    this.chargeUtile = chargeUtile;
}
```

Bien sûr, la charge utile de la requête pourrait être plus complexe, plus structurée, etc. Quant à la réponse :

ReponseSUM.java

```
/*
 * ReponseSUM.java
 */

package ProtocoleSUM;

import java.io.*;

import requetepools.Reponse;

/**
 * @author Vilvens
 */

public class ReponseSUM implements Reponse, Serializable
{
    public static int EMAIL_OK = 201;
    public static int EMAIL_NOT_FOUND = 501;
    public static int KEY_GENERATED = 202;
    public static int WRONG_PASSWORD = 401;

    private int codeRetour;
    private String chargeUtile;

    public ReponseSUM(int c, String chu)
    {
        codeRetour = c; setChargeUtile(chu);
    }

    public int getCode() { return codeRetour; }
    public String getChargeUtile() { return chargeUtile; }
    public void setChargeUtile(String chargeUtile) { this.chargeUtile = chargeUtile; }
}
```

7.9 L'application client

Le client va lui aussi proposer un GUI permettant de choisir le serveur, en termes d'adresse IP et de port, auquel il se connectera quand il voudra envoyer une requête sérialisée. Il lui suffira de lire la réponse par la voie sérialisée analogue.

FenAppClient.java

```
/*
 * FenAppClient.java
 */

package clientpoolthreads;

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

import ProtocoleSUM.*;

/**
 * @author Vilvens
 */

public class FenAppClient extends javax.swing.JFrame
{
    private ObjectInputStream ois;
    private ObjectOutputStream oos;
    private Socket cliSock;

    public FenAppClient() { initComponents(); }
    private void initComponents() { ... }

    private void BEnvoyerActionPerformed (java.awt.event.ActionEvent evt)
    {
        // Construction de la requête
        String chargeUtile = TFRequete.getText();
        RequeteSUM req = null;
        if (RBMail.isSelected())
            req = new RequeteSUM(RequeteSUM.REQUEST_E_MAIL, chargeUtile);
        else req = new RequeteSUM(RequeteSUM.REQUEST_TEMPORARY_KEY,
                                  chargeUtile);

        // Connexion au serveur
        ois=null; oos=null; cliSock = null;
        String adresse = TFAdresseServeur.getText();
        int port = Integer.parseInt(TFPortServeur.getText());
        try
        {
            cliSock = new Socket(adresse, port);
            System.out.println(cliSock.getInetAddress().toString());
        }
    }
}
```

```

        catch (UnknownHostException e)
        { System.err.println("Erreur ! Host non trouvé [" + e + "]); }
        catch (IOException e)
        { System.err.println("Erreur ! Pas de connexion ? [" + e + "]); }

        // Envoie de la requête
        try
        {
            oos = new ObjectOutputStream(cliSock.getOutputStream());
            oos.writeObject(req); oos.flush();
        }
        catch (IOException e)
        { System.err.println("Erreur réseau ? [" + e.getMessage() + "]); }

        // Lecture de la réponse
ReponseSUM rep = null;
        try
        {
            ois = new ObjectInputStream(cliSock.getInputStream());
            rep = (ReponseSUM)ois.readObject();
            System.out.println(" *** Reponse reçue : " + rep.getChargeUtile());
        }
        catch (ClassNotFoundException e)
        { System.out.println("--- erreur sur la classe = " + e.getMessage()); }
        catch (IOException e)
        { System.out.println("--- erreur IO = " + e.getMessage()); }
        LReponse.setText(rep.getChargeUtile());
    }

    public static void main(String args[])
    {
        java.awt.EventQueue.invokeLater(new Runnable()
        {
            public void run()
            {
                new FenAppClient().setVisible(true);
            }
        });
    }

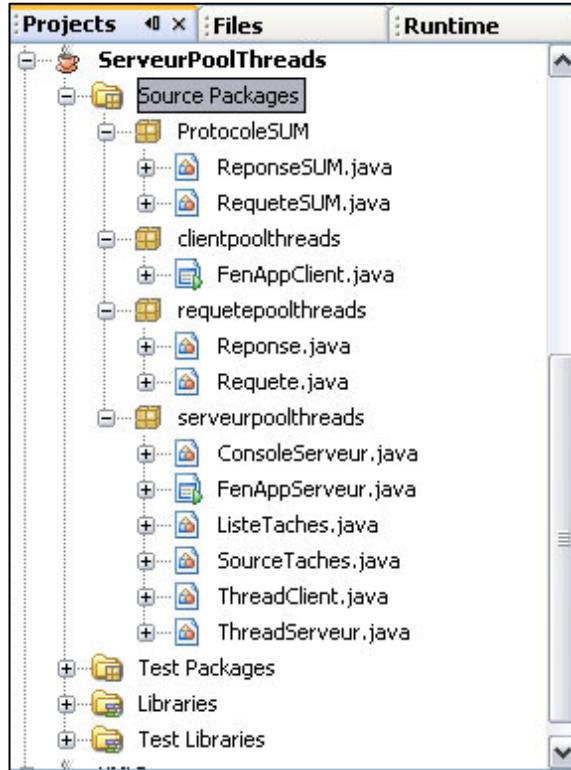
    private javax.swing.JLabel LReponse;
    private javax.swing.JRadioButton RBcle;
    private javax.swing.JRadioButton RBmail;
    private javax.swing.ButtonGroup buttonGroup1;
    private javax.swing.JTextField TFAdresseServeur;
    private javax.swing.JTextField TFPortServeur;
    private javax.swing.JTextField TFRequete;
    private javax.swing.JButton BEnvoyer;
    ...
}

```

Un premier test peut nous faire constater que tout se passe bien avec plusieurs clients : le multithread, la logique du moniteur fournis de tâches, la générativité des requêtes, tout fonctionne correctement.

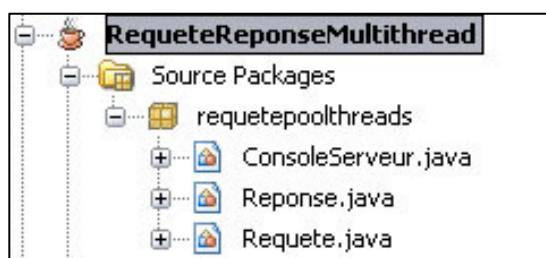
7.10 Le déploiement final du serveur et du client

Tout notre développement a été réalisé dans un seul projet et le jar résultant contient évidemment toutes les classes :

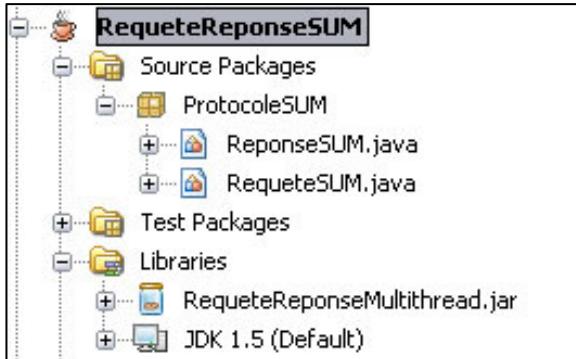


En pratique, nous allons évidemment distinguer le serveur du client, avec des librairies utilisées par l'un et l'autre. Ceci implique une restructuration de nos classes selon 4 projets :

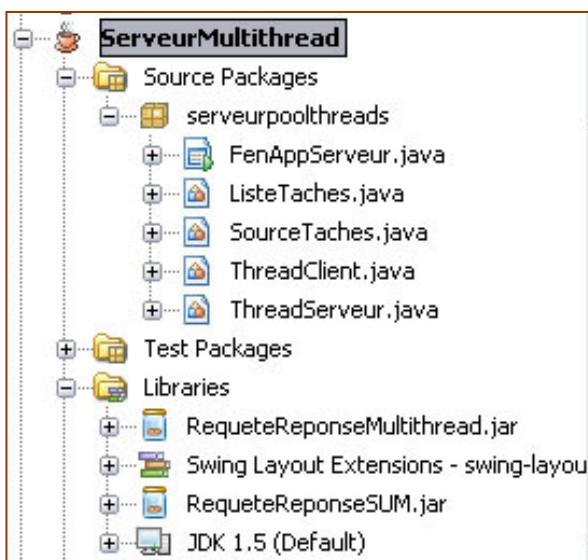
1) RequeteReponseMultithread qui donnera un RequeteReponseMultithread.jar :



2) RequeteReponseSUM qui donnera un RequeteReponseSUM.jar, mais qui utilise aussi RequeteReponseMultithread.jar :



3) ServeurMultithread qui donnera un ServeurMultithread.jar, mais qui utilise aussi l'un des deux jars précédents :



Le serveur pourra donc être déployé n'importe où selon :

```

ServeurMultithread.jar
lib
|
\_\_ RequeteReponseMultithread.jar
  \_\_ RequeteReponseSUM.jar
  \_\_ swing-layout-1.0.jar

```

et lancé par :

```

C:\java-netbeans-application>java -jar ServeurMultithread.jar
Tread client avant get
***** Serveur en attente
Tread client avant get
Tread client avant get
getTache avant wait
getTache avant wait
getTache avant wait
----- Serveur après accept()

```

Requete lue par le serveur, instance de ProtocoleSUM.RequeteSUM

...

Début de traiteRequete : adresse distante = /192.168.1.8:2591

E-Mail trouvé pour Charlet

...

----- Serveur après accept()

Requete lue par le serveur, instance de ProtocoleSUM.RequeteSUM

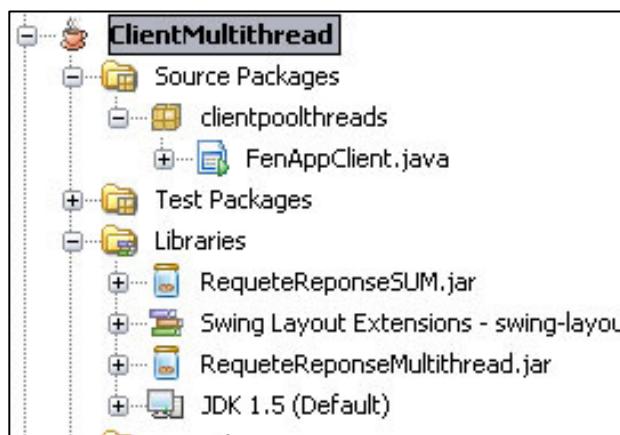
...

Début de traiteRequete : adresse distante = /192.168.1.4:1072

E-Mail trouvé pour Vilvens

...

4) Enfin, ClientMultithread qui donnera un ClientMultithread.jar, mais qui utilise aussi l'un des deux jars précédents :



Le client pourra donc être déployé n'importe où selon :

ClientMultithread.jar

lib

|

_ RequeteReponseMultithread.jar

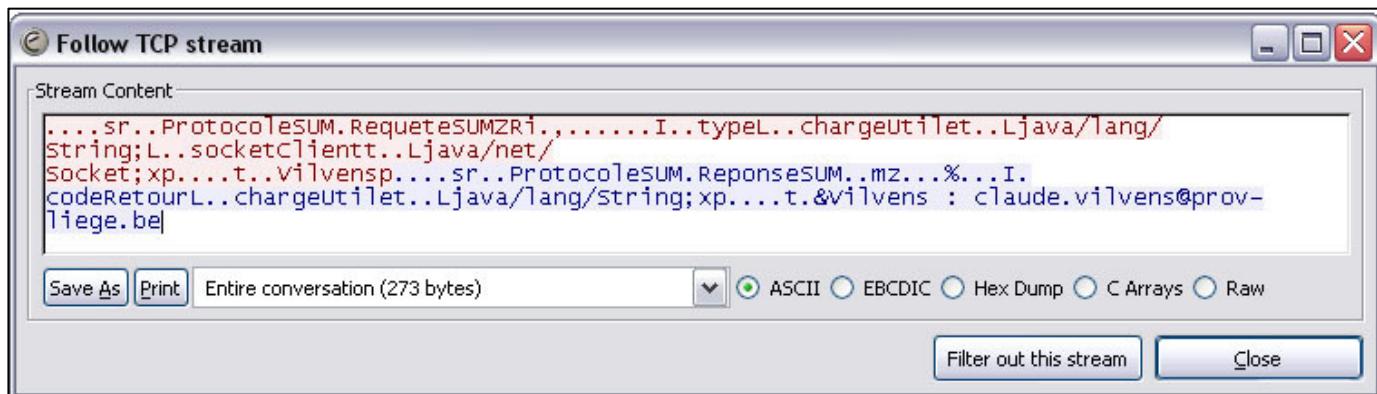
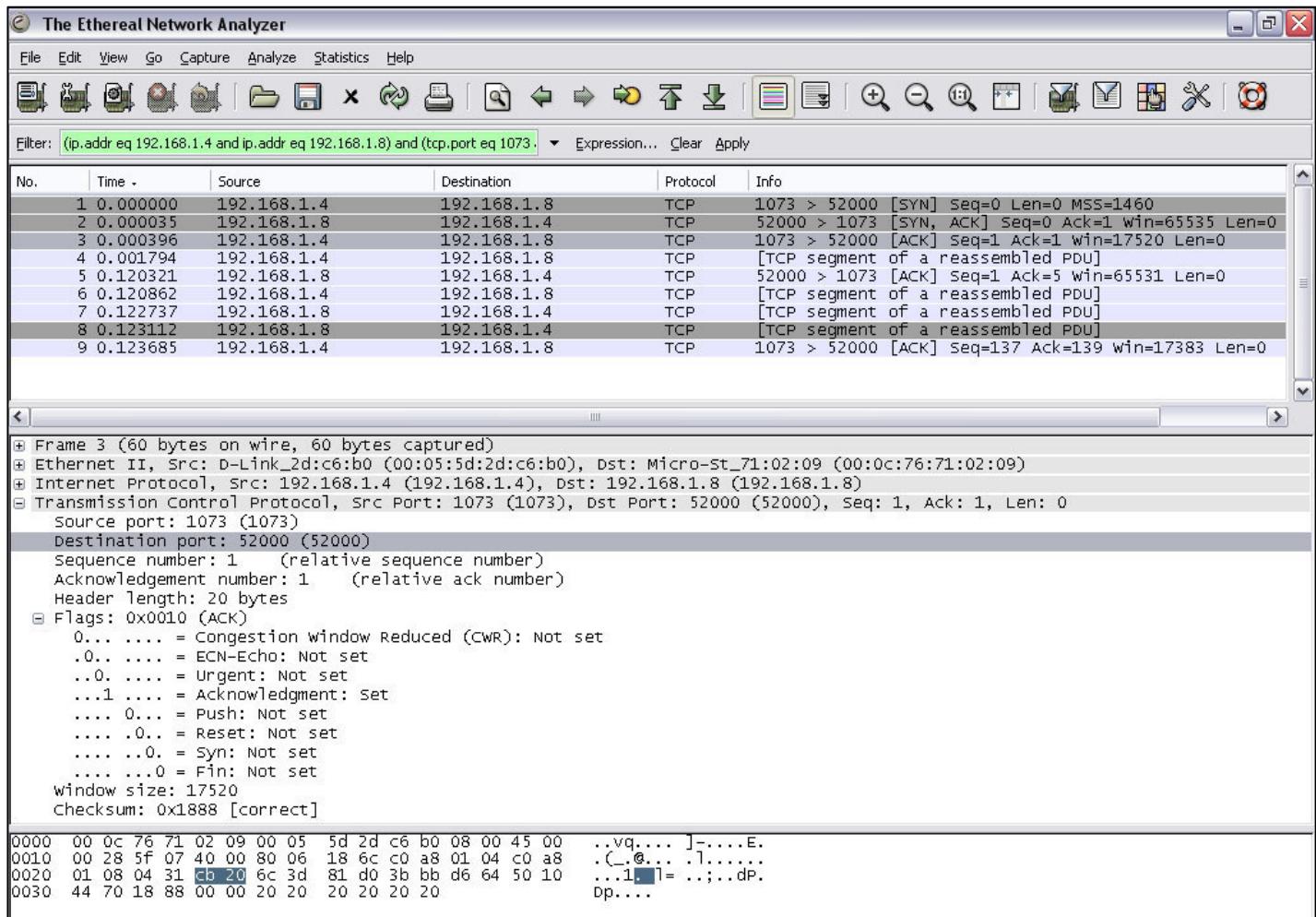
_ RequeteReponseSUM.jar

_swing-layout-1.0.jar

et lancé par :

```
C:\java-netbeans-application>java -jar ClientMultithread.jar
/192.168.1.8
*** Reponse reçue : Charlet : christophe.charlet@prov-liege.be
/192.168.1.8
*** Reponse reçue : fdss : ?@?
/192.168.1.8
*** Reponse reçue : Vilvens : claude.vilvens@prov-liege.be
```

On peut même, pour terminer, visualiser le trafic réseau pour une requête-réponse au moyen d'un sniffer comme Ethereal :



Ouf ! Un petit tour par le frère de TCP ;-) ?

8. Une communication réseau UDP par datagrammes

8.1 Des classes pour UDP

On l'a déjà précisé, le protocole de transport UDP n'est pas orienté connexion, n'est pas fiable et n'assure pas que les données envoyées dans un certain ordre seront reçues selon cet ordre. Ces données constituent des **datagrammes**, c'est-à-dire des "**paquets**" d'information qui se suffisent à eux-mêmes. Ils comportent notamment l'adresse et le port de leur expéditeur, ce qui permet une réponse sans qu'il y ait de connexion permanente. Ces paquets peuvent être routés selon des voies différentes, ce qui explique d'ailleurs le non respect éventuel de l'ordre d'envoi..

On a coutume de comparer les datagrammes aux lettres que l'on confie à La Poste ...

A priori, à quoi peut-il donc servir ? Bien sûr, à traiter des informations pour lesquelles ces caractéristiques ne posent pas de problème, comme par exemple une requête sur tel état du serveur au moment de la réception de la demande.

Le package réseau de Java propose

- ♦ une classe **DatagramPacket** qui matérialise un datagramme reçu ou envoyé; elle possède notamment quatre méthodes

```
public synchronized InetAddress getAddress()  
public synchronized int getPort()  
public synchronized byte[] getData()  
public int getLength()
```

qui permettent de récupérer l'adresse de l'émetteur, le port qu'il a utilisé, les données et leur longueur;

- ♦ une classe **DatagramSocket** qui représente un point d'envoi ou de réception d'un datagramme; elle dispose bien logiquement des méthodes

```
public synchronized void receive(DatagramPacket p) throws IOException
```

Cette méthode est bloquante : elle attend un paquet et bouclera tant que ce paquet hypothétique ne lui sera pas parvenu. Quand ce sera le cas, le paramètre recevra le datagramme envoyé par le client.

```
public void send(DatagramPacket p) throws IOException
```

8.2 Une communication client-serveur par datagramme : le serveur

A titre d'exemple, imaginons un serveur bancaire qui, à chaque requête d'un client, envoie le nom de l'employé de banque qui servira de contact pour les futures transactions entre le titulaire de compte et la banque. On pourrait imaginer que le serveur dispose d'un fichier ou, mieux, d'une base de données, comportant toutes les informations utiles. Pour simplifier, nous nous contenterons cependant d'utiliser un tableau mémoire, qui sera parcouru de manière circulaire.

Notre serveur sera conçu en multithreading, ce qui veut simplement dire que l'on peut imaginer plusieurs threads attendant sur des ports différents. Ces threads seront des instances d'une classe dérivée de Thread : nommons-la très finement ThreadServeur. Le serveur en lui-même se résume donc, par exemple, à :

ServeurBanque.java

```
public class ServeurBanque
{
    public static void main(String args[])
    {
        new ThreadServeur(6000).start();
        new ThreadServeur(7000).start();
    }
}
```

Comment se structure la classe du thread serveur d'employé ? Il aura bien entendu besoin de plusieurs variables membres :

- ♦ un objet DatagramSocket pour la communication :

```
private DatagramSocket socket = null;
```

- ♦ le tableau (statique) des noms des employés.

Son constructeur reçoit comme paramètre le port d'écoute et va créer l'instance de la socket en utilisant le constructeur :

```
public DatagramSocket(int port) throws SocketException
```

qui se connecte au port spécifié sur la machine qui abrite notre serveur. On aurait pu se contenter de

```
public DatagramSocket() throws SocketException
```

qui se connecte à un port disponible sur le serveur. Mais il eut alors fallu que le client soit au courant du choix d'une manière ou l'autre, ce qui n'a rien d'impossible puisque l'objet DatagramPacket reçu peut fournir son port.

Une fois lancé, le thread exécutera sa méthode **run()** sur laquelle repose évidemment tout le travail. Celle-ci va successivement :

- ♦ s'assurer que la connexion existe toujours;
- ♦ se mettre en attente d'un datagramme au moyen de la méthode **receive()**, déjà évoquée; l'objet DatagramPacket utilisé utilisera le constructeur :

```
public DatagramPacket (byte[] buf, int length)
```

- ♦ une fois le paquet reçu, en extraire l'adresse et le port de l'expéditeur (ce sera utile pour lui répondre);
- ♦ rechercher le nom de l'employé auquel on est parvenu dans le tableau statique;
- ♦ construire un datagramme avec la réponse, en utilisant l'autre constructeur :

public **DatagramPacket**(byte ibuf[],int ilength, InetAddress iaddr, int iport)

qui permet de tenir compte de l'adresse et du port du destinataire du paquet;

- ◆ envoyer la réponse au moyen de la méthode send.

Au total, notre classe de thread serveur s'écrira :

ThreadServeur.java

```
/*
 * ThreadServeur.java
 * Created on 10 mai 2004, 16:50
 */

/**
 *
 * @author Vilvens
 */

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

public class ThreadServeur extends Thread
{
    private DatagramSocket socket = null;
    static final int nbregenies = 5;
    static String[] genies = new String[nbregenies];
    static
    {
        genies[0] = "Charlet Terminator";
        genies[1] = "Wagner Signalor";
        genies[2] = "Vilvens Exterminator";
        genies[3] = "Merce Unixator";
        genies[4] = "Madani Investigator";
    }
    ThreadServeur (int port)
    {
        super("ThreadServeur");
        try
        {
            socket = new DatagramSocket(port);
            System.out.println("Je vous attend sur le port " + socket.getLocalPort());
        }
        catch(SocketException e)
        {
            System.err.println("Impossible de créer une socket UDP");
        }
    }
}
```

```

public void run()
{
    if (socket == null) return;
    int pos=0;
    while (true)
    {
        try
        {
            byte[] buf = new byte[256]; int bufLen=256;
            DatagramPacket paquet;      InetAddress adresse;
            int port;
            String chaine = null;

            // réception d'une requête
            paquet = new DatagramPacket(buf, 256);
            socket.receive(paquet);
            adresse = paquet.getAddress();      port = paquet.getPort();
            System.out.println("Requête reçue de " + adresse + " - port: " + port);

            // recherche d'un nom
            chaine = genies[pos++];      if (pos>4) pos=0;

            // envoi de la réponse
            buf = chaine.getBytes();bufLen = buf.length;
            System.out.println("Employé = " + chaine);
            System.out.println("longueur envoyée = " + bufLen);
            paquet = new DatagramPacket(buf,bufLen, adresse, port);
            socket.send(paquet);
        }
        catch (IOException e)
        {
            System.err.println("Problème : " +e.getMessage());
        }
    }
}

```

On remarquera l'utilisation de la méthode :

```
public byte[] getBytes()
```

qui retourne la conversion d'un objet String en une suite de bytes selon le système d'encodage des caractères en vigueur sous le système utilisé.

8.3 Une communication client-serveur par datagramme : le client

Le client va, en quelque sorte, réaliser les opérations "miroir" du serveur, si ce n'est qu'il ne va pas utiliser un thread :

- ◆ instantiation d'une socket sur l'adresse et le port visé – celui-ci sera passé sur la ligne de commande;
- ◆ envoi d'un paquet requête;
- ◆ attente de la réponse;
- ◆ affichage de celle-ci.

Pour spécifier l'adresse du serveur, on utilisera la méthode de InetAddress :

```
public static InetAddress getByName(String host) throws UnknownHostException
```

qui permet d'obtenir un objet instanciant la classe adresse InetAddress à partir du nom de la machine ou de son adresse IP sous forme de chaîne de caractères.

ClientBanque.java

```
/*
 * ClientBanque.java
 *
 * Created on 10 mai 2004, 17:00
 */

/**
 *
 * @author Vilvens
 */

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

public class ClientBanque
{
    static public final int PORT_PAR_DEFAULT = 6000;
    public static void main(String args[])
    {
        System.out.println ("Démarrage !");
        DatagramSocket socket = null;
        byte[] buf, bufR = new byte[256];
        DatagramPacket paquet;
        int port;      // port d'écoute du serveur
        if (args.length>0) port = Integer.parseInt(args[0]);
        else port = PORT_PAR_DEFAULT;
        try
        {
            for (int i=0; i<10; i++) // une petite boucle
            {
                InetAddress adresse = InetAddress.getByName("claude");
            }
        }
    }
}
```

```
String chaine = "Je voudrais un contact ...";  
  
// creation d'une socket  
socket = new DatagramSocket();  
  
// envoi d'une requete  
buf = chaine.getBytes();  
paquet = new DatagramPacket(buf, buf.length, adresse, port);  
socket.send(paquet);  
System.out.println("Requete envoyée");  
  
// attente de la réponse  
paquet = new DatagramPacket(bufR, 256);  
socket.receive(paquet);  
String chaineReçue = new String (paquet.getData());  
System.out.println("Nombre de bytes reçus : " + paquet.getLength());  
String nomEmployé = chaineReçue.substring (0, paquet.getLength());  
System.out.println( "Nom de l'employé = " + nomEmployé);  
}  
}  
catch (UnknownHostException e)  
{  
    System.err.println("Problème host : " +e.getMessage());  
}  
catch(SocketException e)  
{  
    System.err.println("Problème socket : " +e.getMessage());  
}  
catch(IOException e)  
{  
    System.err.println("Problème IO : " +e.getMessage());  
}
```

On remarquera encore l'utilisation du constructeur

public **String**(**bytes**[] byte)

qui construit un objet String à partir d'un tableau de bytes en utilisant le système d'encodage des caractères utilisé par la plate-forme.

ainsi que de la méthode d'extraction de sous-chaîne :
public String **substring**(int beginIndex, int endIndex)

pour ne conserver que les bytes constituant la réponse effective. On obtient comme résultat :

le serveur :

Je vous attend sur le port 6000
Je vous attend sur le port 7000

Requete reçue de /192.168.2.1 - port: 1053
Employé = Charlet Terminator
longueur envoyée = 18
Requete reçue de /192.168.2.1 - port: 1054
Employé = Wagner Signalor
longueur envoyée = 17
Requete reçue de /192.168.2.1 - port: 1055
Employé = Vilvens Exterminator
longueur envoyée = 20
Requete reçue de /192.168.2.1 - port: 1056
Employé = Merce Unixator
longueur envoyée = 14
Requete reçue de /192.168.2.1 - port: 1057
Employé = Madani Investigator
longueur envoyée = 19
Requete reçue de /192.168.2.1 - port: 1058
Employé = Charlet Terminator
longueur envoyée = 18
Requete reçue de /192.168.2.1 - port: 1059
Employé = Wagner Signalor
longueur envoyée = 17
Requete reçue de /192.168.2.1 - port: 1060
Employé = Vilvens Exterminator
longueur envoyée = 20
Requete reçue de /192.168.2.1 - port: 1061
Employé = Merce Unixator
longueur envoyée = 14
Requete reçue de /192.168.2.1 - port: 1062
Employé = Madani Investigator
longueur envoyée = 19

le client :

Démarrage !
Requete envoyée
Nombre de bytes reçus : 18
Nom de l'employé = Charlet Terminator
Requete envoyée
Nombre de bytes reçus : 17
Nom de l'employé = Wagner Signalor
Requete envoyée
Nombre de bytes reçus : 20
Nom de l'employé = Vilvens Exterminator
Requete envoyée
Nombre de bytes reçus : 14
Nom de l'employé = Merce Unixator
Requete envoyée
Nombre de bytes reçus : 19
Nom de l'employé = Madani Investigator
Requete envoyée
Nombre de bytes reçus : 18

```
Nom de l'employé = Charlet Terminator
Requête envoyée
Nombre de bytes reçus : 17
Nom de l'employé = Wagner Signalor
Requête envoyée
Nombre de bytes reçus : 20
Nom de l'employé = Vilvens Exterminator
Requête envoyée
Nombre de bytes reçus : 14
Nom de l'employé = Merce Unixator
Requête envoyée
Nombre de bytes reçus : 19
Nom de l'employé = Madani Investigator
```

Il faudrait évidemment compléter en permettant au serveur de terminer proprement ...

9. La communication réseau basée sur HTTP

9.1 Les URLs

Les différentes machines connectées sur un réseau sont connues par une adresse IP qui est une identification unique, similaire au numéro d'une maison dans une rue. Cette adresse se compose de 4 nombres (de 0 à 255) séparés par des points. Pour être exact, elle s'accompagne d'un masque déterminant la nature du sous réseau, de la forme 255.255.255.0, 255.255.0.0, etc

La manipulation de telles adresses brutes n'est pas forcément des plus aisées. Aussi, sur Internet, on utilise plutôt un alias de l'adressage IP connu sous le nom d'**URL** (Uniform Ressource Locator). Une URL comporte plusieurs composantes :

```
<nom du protocole>://<nom de la machine hôte>[::<numéro de port>]/<chemin d'accès>
```

Le protocole est **http** (hypertext transfer protocol) dans le cas de transfert de pages WEB. Pour le transfert de fichiers, ce protocole est **ftp** (file transfer protocol). Et pour la messagerie électronique, le protocole est **mailto** (mailing to). Enfin, même pour atteindre un fichier local (c'est-à-dire se trouvant sur la machine hébergeant le browser), il faudra citer le protocole **file**.

Le nom de la machine hôte (**hostname**), c'est-à-dire le serveur auquel on se connecte, est le nom sous lequel le serveur est connu sur Internet. C'est un moyen plus commode de le désigner que d'utiliser son adresse IP. Le nom logique commence souvent par www, mais ce n'est pas une obligation. A remarquer qu'une URL peut désigner en fait toute ressource adressable. Pour rappel, ce **nom logique**, car c'en est bien un, est en fait formé de trois composantes correspondant à une organisation hiérarchique à trois niveaux :

- ◆ le niveau supérieur identifie un **domaine** qui correspond
 - ⇒ soit au domaine d'activité au niveau international : les sigles sont
 - ◊ .com [commercial],
 - ◊ .org [organisations à but non lucratif],
 - ◊ .int [organisations internationales],
 - ◊ .net [fournisseurs de services réseau];

- ⇒ soit à des réseaux dépendant de l'administration américaine : les sigles sont
 - ◊ .gov [gouvernement et administrations],
 - ◊ .mil [militaire],
 - ◊ .edu [éducatif - universités];
 - ⇒ soit au pays pour les réseaux nationaux : on utilise les codes ISO à deux lettres du pays (.be, .fr, .uk, .us, .jp, ...).
-
- ◆ le niveau intermédiaire identifie un **sous-domaine** qui correspond à une entité économique, par exemple microsoft, cern, nasa, **prov-liege**, ...
 - ◆ le troisième niveau désigne **la machine elle-même**, par un nom quelconque : www, web, ulyss, ...

On évite ainsi de spécifier le serveur au moyen de son adresse réseau pure et dure, c'est-à-dire de son adresse TCP/IP.

Le chemin d'accès désigne l'emplacement de ce que l'on cherche sur le serveur. Sans autre précision, on aboutira, dans le cas du protocole http, à la home page du serveur. On accède au serveur du réseau de l'enseignement de la Province de Liège par l'URL
http://www.prov-liege.be

Il se peut que l'URL comporte un numéro de port. C'est le cas lorsqu'il faut se connecter à un port autre que le port par défaut (qui est, en général, 80 pour http).

Remarques

1) Chaque domaine possède un serveur de nom ou **DNS** (Domain Name Server). Il est chargé d'effectuer la correspondance entre les noms logiques et les adresses numériques (celles-ci sont les seules qui sont utilisables dans les communications).

2) Dans le cas d'un fichier local, l'URL est un peu particulière :

- ◆ il y a trois slash après le nom du protocole (soit file:);
- ◆ une unité de disque locale est désignée par une lettre suivie de ‘l’ (ASCII 124) au lieu de ‘:’, qui a un autre sens pour http.

ex : **file:///C:/htmpar/home-eplnet.htm**

9.2 La classe URL

Au **niveau applicatif**, Java propose, au sein de son package net, la classe **URL** qui matérialise bien sûr le concept Internet correspondant. Une telle classe permet de communiquer sur le réseau au niveau de la couche application (et pas transport) dans le but d'accéder à une ressource Internet. Les constructeurs ont des prototypes bien logiques :

- ◆ URL(String) ; // l'URL est passée sous forme de chaîne de caractères
- ◆ URL(String, String, String) // protocole, hôte, fichier
- ◆ URL(String, String, int, String) // protocole, hôte, numéro de port, fichier
- ◆ URL(URL, String) // l'URL de base est complétée d'un chemin relatif ou // d'un signet (#xxx) dans la page HTML (protocole http)

En cas d'erreur (protocole inconnu, référence nulle, etc), c'est un objet de la classe d'exception **MalformedURLException** qui est lancé. Le découpage d'une URL se réalise au moyen des méthodes attendues :

- ◆ public String **getProtocol()**
- ◆ public String **getHost()**
- ◆ public int **getPort()**
- ◆ public String **getFile()**
- ◆ public String **getRef()**

On ne fera pas l'injure au lecteur de préciser les valeurs renvoyées par ces méthodes. Signalons cependant que la dernière méthode renvoie l'URL de base, encore désignée par "référence" (*anchor*); elle ne fonctionne correctement que si on a utilisé un constructeur qui réclame cette référence (du moins dans le JDK 1.0).

Une méthode intéressante d'URL est :

```
public final InputStream openStream() throws IOException
```

Comme son nom l'indique, cette méthode ouvre une connexion sur l'URL considérée et **fournit un objet InputStream permettant une lecture depuis cette connexion**. Il suffit dès lors de se donner, par exemple, un objet DataInputStream construit sur cet objet pour pouvoir lire le contenu de l'élément visé par l'URL. L'exemple suivant permet de se connecter sur une machine UNIX appelée dec01 :

URL01.java

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

public class URL01
{
    public static void main(String args[])
    {
        try
        {
            URL eplnet = new URL("http://dec01.inpres.epl.prov-liege.be:85");
            System.out.println("1. protocole = " + eplnet.getProtocol());
            DataInputStream dis = new DataInputStream(eplnet.openStream());
            String ligne;
            while ( (ligne=dis.readLine()) != null ) System.out.println(ligne);
            dis.close();
        }
        catch (MalformedURLException e)
        {
            System.err.println("Erreur ! URL non trouvée [" + e + "]");
        }
        catch (IOException e)
        {
            System.err.println("Erreur ! ? [" + e + "]");
        }
    }
}
```

On remarquera les deux exceptions traitées, l'une pour le réseau et l'autre pour les E/S. Le résultat sera du type :

```
1. protocole = http
<HTML>
<HEAD><TITLE>index de webusr</TITLE>
</HEAD>
<BODY> Bonjour à tous ...
</BODY>
</HTML>
```

si un serveur HTTP était à l'écoute à l'URL indiquée sur le port 85, avec une page HTML index rudimentaire. A remarquer que l'en-tête HTTP n'apparaît pas dans ce qu'on lit : c'est logique, puisque **la communication réseau est ici supportée par le protocole HTTP**. Cet en-tête apparaîtrait en toutes lettres si la connexion sa faisait avec une classe Socket (couche transport - voir plus loin). D'ailleurs, dans ce cas, la requête à envoyer sur le flux devrait contenir le header typique de http:

```
GET + <nom fichier> + " HTTP/1.0\r\n" + "User-Agent : ..." + "Accept: text/*\r\n"
```

9.3 La classe URLConnection

En fait, `openStream()` est une abréviation de l'appel `openConnection().getInputStream()`.

La méthode de la classe URL

```
public URLConnection openConnection() throws IOException
```

renvoie un objet instanciant la classe ***URLConnection*** qui matérialise la connexion à l'objet distant désigné par l'URL. On s'en doute, cette classe représente la classe de base de toutes les classes représentant une connexion entre l'application et une ressource localisée par une URL. Le seul constructeur est

```
URLConnection(URL)
```

La méthode

```
public InputStream getInputStream() throws IOException
```

fournit évidemment un flux permettant de lire depuis la connexion. Le programme précédent peut donc être réécrit comme suit :

URL01.java (bis)

```

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

public class URL01
{
    public static void main(String args[])
    {
        try
        {
            URL eplnet = new URL("http://dec01.inpres.epl.prov-liege.be:85");

            System.out.println("2. protocole = "+ eplnet.getProtocol());
            URLConnection eplnetConnection = eplnet.openConnection();

            String ligne;
            DataInputStream dis = new DataInputStream(eplnetConnection.getInputStream());
            while ( (ligne=dis.readLine()) != null ) System.out.println(ligne);
            dis.close();
        }
        catch (MalformedURLException e)
        {
            System.err.println("Erreur ! URL non trouvée [" + e + "]");
        }
        catch (IOException e)
        {
            System.err.println("Erreur ! ? [" + e + "]");
        }
    }
}

```

Inutile de dire que l'on peut aussi ouvrir un **DataOutputStream** sur le flux, ce qui permet d'écrire, par exemple dans un formulaire communicant, par exemple, avec une servlet ou un listener Oracle ... Plus précisément, l'objet **URLConnection** peut fournir un flux d'écriture (au lieu d'un flux de lecture) si l'on spécifie la volonté décrire sur le flux en utilisant la méthode

public void setDoOutput(boolean dooutput)

Cette méthode positionne une variable membre **protected** :

protected boolean doOutput

dont le rôle est bien clair. Elle possède une sœur **doInput** au rôle semblable, mais fixé à true par défaut. Il reste à se connecter par

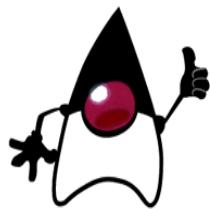
public abstract void connect() throws IOException

qui réalise la connexion à ce qui est référencé par l'URL si cette connexion n'existe pas encore.

Remarque

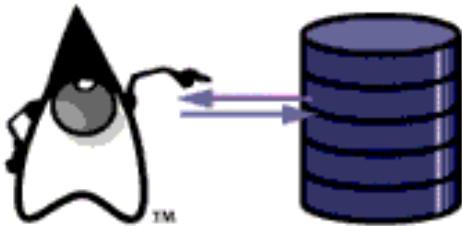
Dans le cas où la plate-forme hôte du client utilise un *proxy*, il faut le spécifier à l'interpréteur.
Sur la ligne de commande, on écrira :

```
java -DproxySet=true -DproxyHost=<nom du proxy> URL01.class
```



Nous reviendrons aux communications réseaux basées HTTP dans le ([très] gros) chapitre consacré aux servlets. Mais ce qui motive souvent l'accès à un serveur est la demande d'informations qu'il possède dans une base de données. Comment programmer des requêtes sur ces bases, le plus souvent relationnelles ?

XII. JDBC et l'accès aux bases de données



La grande histoire véritable est celle des inventions ...

(R. Queneau; Bâtons, chiffres et lettres)

1. Présentation

Contrairement à ce que l'on croit souvent, **JDBC**, produit par Javasoft, n'est pas, paraît-il, un acronyme, comme par exemple ODBC. Cependant, on a coutume de considérer que le sigle abrège "Java DataBase Connectivity", ce qui a le mérite de spécifier d'emblée de quoi il s'agit ...

JDBC est donc bien un ensemble d'APIs permettant d'accéder, au moyen d'instructions SQL, aux informations stockées dans une base de données. Dans une large mesure, la plupart des SGBD et gestionnaires de fichiers courants sont gérés, si bien que l'on peut considérer que le code écrit avec JDBC est indépendant de ceux-ci. Comme Java produit un bytecode portable sur les machines courantes, on peut considérer qu'une application Java-JDBC présente un large spectre de portabilité.

Le dernière version en date est **JDBC 4.0** (packages `java.sql` et `javax.sql`) et est incluse dans le JDK 1.6. Elle comporte les concepts avancés des bases de données relationnelles, comme les types non basiques (BLOB, etc), les pools de connexions aux bases et la compatibilité ANSI SQL-3.

La démarche d'utilisation (inspirée à l'époque de l'ODBC de Microsoft) comporte :

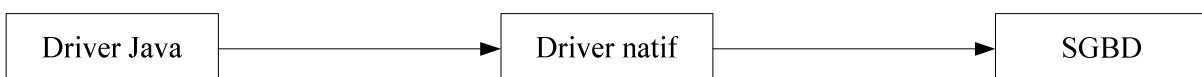
- ◆ une connexion à la base de données visée;
- ◆ l'envoi de commandes SQL;
- ◆ la réception, pour traitement, des résultats des requêtes.

On constate donc que, pour utiliser JDBC, une certaine connaissance de SQL est nécessaire. En ce sens, on peut dire qu'il s'agit d'un interface "bas niveau", par opposition à un "haut niveau" qui permettrait d'interroger la base de manière plus intuitive (c'est ce que propose la technologie **JPA** [Java Persistence API] – voir annexe 3 en fin d'ouvrage).

2. Les drivers et les modèles possibles

Bien sûr, c'est le rôle d'un "driver JDBC" ("pilote JDBC en français) de communiquer avec le SGBD visé : il sert de pont entre l'application Java et ce SGBD. C'est le rôle d'un DriverManager de localiser les différents drivers possibles (nous allons y revenir). On en distingue de 4 types :

- ◆ **type I** : Le driver Java mappe simplement un driver natif (donc par exemple écrit en C comme ODBC) :



- ◆ **type II** : Le driver est partiellement écrit en Java partiellement avec du code natif et utilise en fait une librairie native :



- ◆ **type III** : Le driver est cette fois "full Java" et il utilise un middleware qui réalise l'accès au SGBD; la communication driver-middleware est donc indépendante du protocole d'accès à la base à travers le SGBD :



- ◆ **type IV** : Le driver, entièrement écrit en Java, implémente le protocole d'accès à la base à travers le SGBD; il est donc "propriétaire" dans le sens qu'il est dédié à un type de SGBD donné :



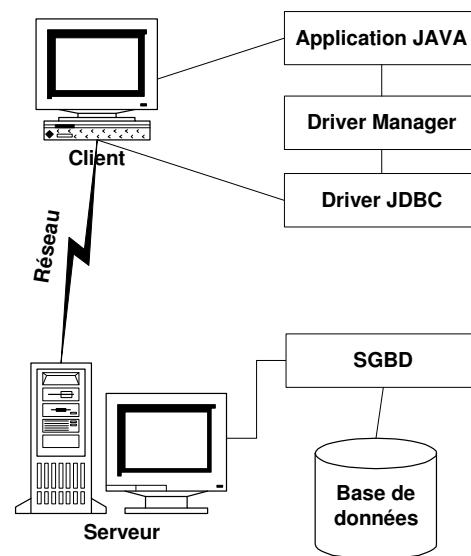
Le nombre de drivers JDBC disponibles est impressionnant : un site comme <http://mindprod.com/jgloss/jdbcvendors.html> permet de consulter une liste de "vendors" avec le type de driver fourni et un lien pour télécharger. Citons (avec le nom sont fournies la chaîne de connexion (voir plus loin) et la classe driver) :

JDBC-ODBC Bridge
jdbc:odbc:<DB>
sun.jdbc.odbc.JdbcOdbcDriver
Oracle Thin
jdbc:oracle:thin:@<HOST>:<PORT>:<SID>
oracle.jdbc.driver.OracleDriver
Oracle OCI 8i
jdbc:oracle:oci8:@<SID>
oracle.jdbc.driver.OracleDriver
Oracle OCI 9i
jdbc:oracle:oci:@<SID>
oracle.jdbc.driver.OracleDriver
Microsoft SQL Server
jdbc:weblogic:mssqlserver4:<DB>@<HOST>:<PORT>
weblogic.jdbc.mssqlserver4.Driver
Microsoft SQL Server 2000 (Microsoft Driver)
jdbc:microsoft:sqlserver://<HOST>:<PORT>[;DatabaseName=<DB>]
com.microsoft.sqlserver.jdbc.SQLServerDriver
Microsoft SQL Server (JT Turbo Driver)
jdbc:JT Turbo://<HOST>:<PORT>/<DB>
com.ashna.jturbo.driver.Driver
CSV Files
jdbc:relique:csv:<FILE>, <Properties FILE>
org.relique.jdbc.csv.CsvDriver

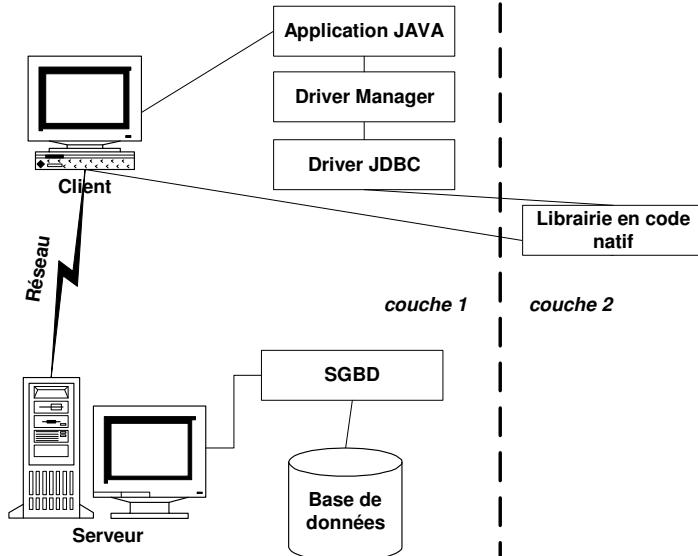
MySQL (MM.MySQL Driver) jdbc:mysql://<HOST>:<PORT>/<DB> org.gjt.mm.mysql.Driver
PostgreSQL (v7.0 and later) jdbc:postgresql://<HOST>:<PORT>/<DB> org.postgresql.Driver
Sybase (jConnect 5.2) jdbc:sybase:Tds:<HOST>:<PORT> com.sybase.jdbc2.jdbc.SybDriver

L'accès aux bases de données par JDBC peut se faire selon trois modèles que l'on désigne en anglais par "*one-tier*", "*two-tier*" et "*three-tier*" :

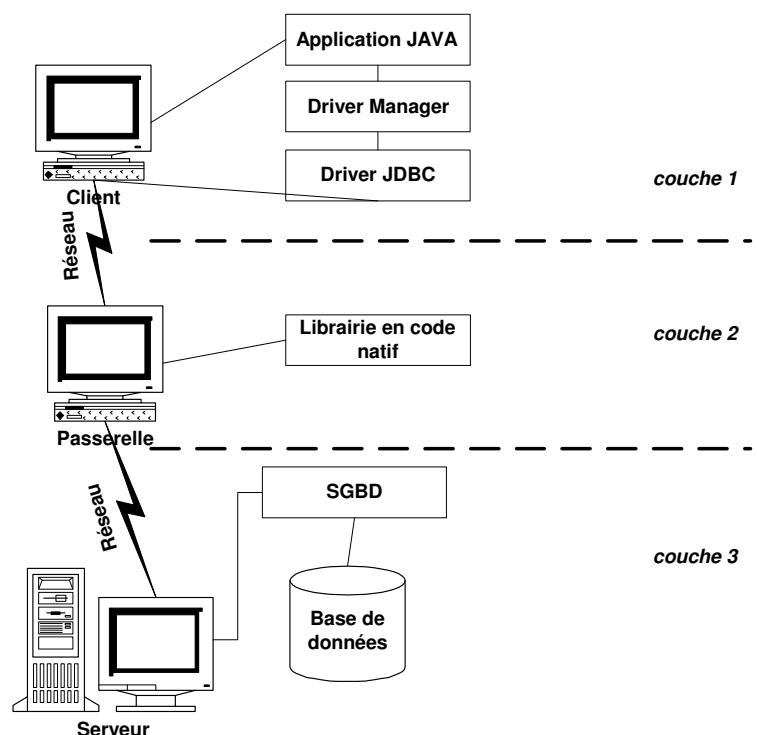
♦ **le modèle client-serveur classique à un niveau :**



- ◆ **le modèle client-serveur à deux niveaux :**



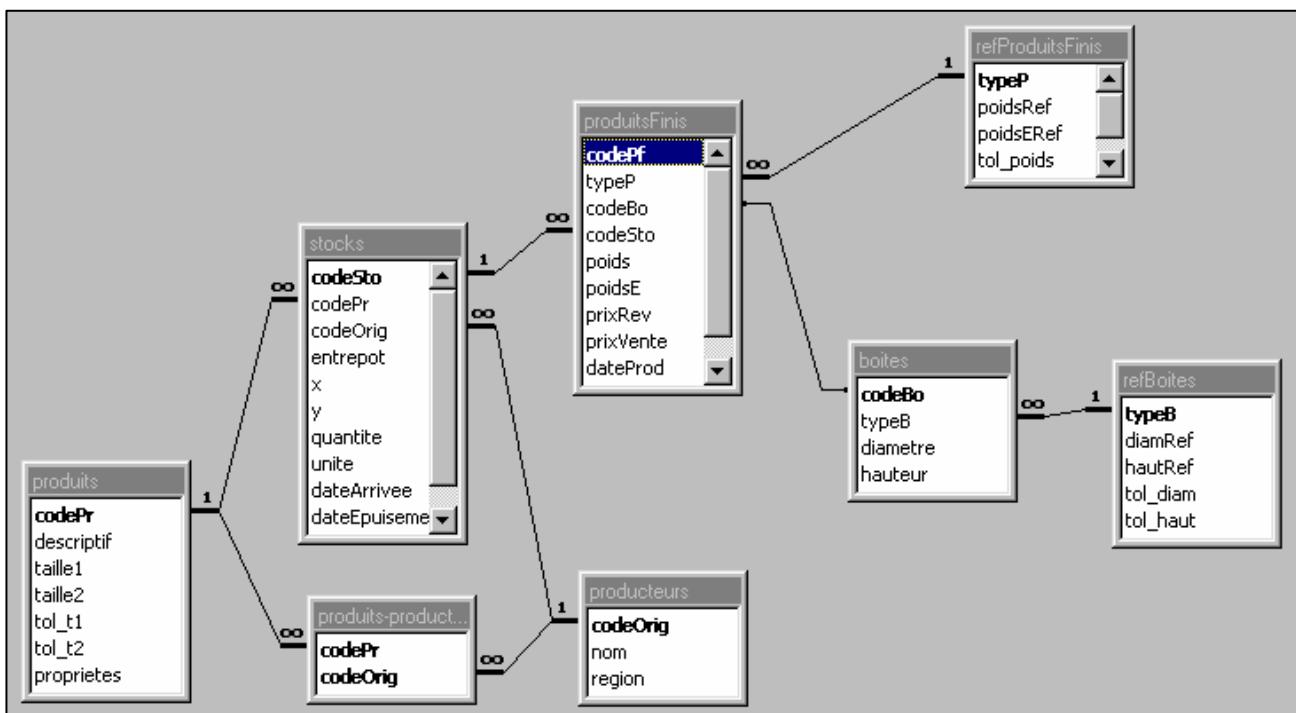
♦ le modèle client-passerelle-serveur
à trois niveaux :



3. Etablir une connexion à une base de données

3.1 Un contexte exemple

Considérons par exemple une base de données dénommée "marie-inpres", gérée au moyen de MS-Access. L'objectif de cette base est de gérer les productions d'une entreprise de fabrication de conserves de légumes et de fruits. La base de données possède la structure suivante :



Notre souhait est donc d'aller chercher des informations dans cette base de données depuis une application Java. Dans ce cas particulier, le driver est le pont vers ODBC. On supposera donc que le driver ODBC de MS-ACCESS a été installé et que la base de données constitue une source de données (appelons-la "marie") connue du driver ODBC.

Nous aurons inévitablement besoin des classes du package `java.sql` et notre programme débutera donc par : `import java.sql.*;`

3.2 La classe Class et le chargement du pilote

La première étape est de **localiser le pilote** (le driver) correspondant au SGBD utilisé pour la base visée. Ici, il faut passer par ODBC et le pilote est donc le **JdbcOdbcDriver**. Les drivers JDBC nécessaires sont désignés selon la syntaxe habituelle des classes et packages. Ainsi, la classe nécessaire est ici :

sun.jdbc.odbc.JdbcOdbcDriver

Pour un autre SGBD ou logiciel de gestion de données, il faudra utiliser évidemment une autre classe, dont le nom exact sera fourni dans la documentation du driver JDBC fourni. Ainsi, pour le driver `jdbc` correspondant au SGBD MySQL sous Linux, la classe nécessaire est `com.mysql.jdbc.driver` (ou `org.gjt.mm.mysql.Driver`), tandis que pour Oracle, il s'agira de `oracle.jdbc.driver.OracleDriver`.

Le chargement de la classe du pilote se fait en utilisant une méthode de la classe **Class**. Cette classe, comme son nom l'indique, sert à représenter l'ensemble des renseignements sur une classe, comme son nom ou ses différentes méthodes; on trouve ainsi les méthodes suivantes :

```
public native String getName();
public Field[] getDeclaredFields() throws SecurityException;
public Method[] getDeclaredMethods() throws SecurityException;
public native Class[] getInterfaces();
```

les classes `Field` et `Method` jouant bien le rôle que l'on devine. Nous reviendrons sur cette classe dans un chapitre ultérieur. Pour notre présent propos, c'est la méthode

```
public static native Class forName(String className) throws ClassNotFoundException;
```

qui va nous intéresser. En effet, cette méthode de classe est invoquée avec le nom d'une classe que l'on recherche : la méthode, au moment de l'exécution, va **localiser** la déclaration de la classe, en **charger la définition** (donc aussi *ses blocs statiques qui constituent le driver*) et **ajouter** cette définition à l'environnement d'exécution de l'application courante. Si l'opération s'est bien déroulée, la méthode renvoie un objet `Class` qui renseigne sur ce que l'on trouvé – en cas d'échec, c'est l'exception `ClassNotFoundException` qui est lancée. Ce mécanisme dynamique est encore appelé l"**"introspection"**".

Comme notre objectif est ici de retrouver le pilote `sun.jdbc.odbc.JdbcOdbcDriver`, nous allons demander à cette méthode miracle de le faire pour nous :

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

Cette méthode réalise donc la recherche de la classe du driver demandé mais enregistre aussi implicitement celui-ci dans la liste des drivers disponibles. C'est ce que ferait la méthode :

```
public static synchronized void registerDriver(Driver driver) throws SQLException
```

qui permet à un objet de la classe DriverManager de connaître l'existence du nouveau driver.

3.3 La classe DriverManager

Cette classe est en fait la partie visible de la gestion même de JDBC. L'action de ses méthodes se situe entre les utilisateurs et les drivers JDBC : ainsi, c'est l'une de ses méthodes, `getConnection()`, qui établit une connexion entre base de données visée et le driver ad hoc. Pour cela, elle gère également la liste des drivers disponibles. On dit encore qu'ils sont "enregistrés" (`registered`).

Toutes les méthodes de DriverManager sont des méthodes de classe – autrement dit, elles sont statiques ! Ceci implique qu'elles sont utilisées sans instanciation de la classe. D'ailleurs, le seul constructeur est privé, empêchant ainsi toute instanciation directe. La seule méthode utilisée couramment est :

```
public static synchronized Connection getConnection(String url, String user,  
String password) throws SQLException;
```

qui permet de spécifier la base visée, avec l'identification de l'utilisateur et son mot de passe. Elle existe d'ailleurs en versions polymorphes, notamment avec le seul paramètre définissant la base.

A titre documentaire, citons d'autres méthodes de DriverManager :

◆ `public static int getLoginTimeout();`

Elle fournit, en secondes, le temps maximum d'attente d'un driver lorsqu'il tente de se connecter à une base de données

◆ `public static void setLoginTimeout(int seconds)`

Elle fixe ce temps.

◆ `public static void setLogStream(PrintStream out);`

Elle définit le flux (un `PrintStream`) utilisé comme fichier de log par le Driver Manager ainsi que tous les drivers.

Etc.

3.4 Obtenir une connexion

Le driver étant chargé, il faut obtenir une connexion sur la base de données visée. Ceci se fait, comme évoqué ci-dessus, en utilisant la méthode statique `getConnection()` de la classe DriverManager. Cette méthode tente de sélectionner le driver approprié parmi ceux qui ont été enregistrés puis, en cas de succès, fournit un objet qui implémente l'interface `Connection`, matérialisant la connexion demandée. La base de données visée est désignée par son URL, c'est-à-dire une chaîne de caractères de la forme :

`jdbc:<nom du sous-protocole>:<alias du nom de la base pour ce protocole>`

Dans notre cas, le "sous-protocole" est odbc et l'alias est le nom de la source de données :

```
Connection con = DriverManager.getConnection("jdbc:odbc:marie","","");
```

ou même

```
Connection con = DriverManager.getConnection("jdbc:odbc:marie");
```

Mais le sous-protocole pourrait être aussi "mysql" avec une adresse IP (ou le nom DNS correspondant), le port du serveur et le nom de la base visée ("`jdbc:mysql://localhost:3306/DB_Food`", "", "") ou encore "oracle" ("`jdbc:oracle:thin@localhost:1521:DB_secrets`") (un service de nommage réseau [*network name service*] peut être chargé de résoudre le nom d'alias utilisé),.

Remarque

Il n'est pas possible d'utiliser le pont JDBC -ODBC au sein d'une applet. En effet, une applet ne peut exécuter un programme local : or, ODBC est en code natif ... Par conséquent, alors qu'un driver JDBC entièrement Java serait parfaitement utilisable, il n'en est pas de même du "driver" JDBC-ODBC pour les raisons habituelles de sécurité.

4. L'interface Connection

Comme son nom l'indique, un objet qui implémente cet interface matérialise une connexion avec une base de données, ce qui comprend également les commandes SQL en cours et leurs résultats. Il serait donc plus approprié de parler de "session". Une application peut avoir plusieurs connexions avec une même bases de données ou avec des bases de données différentes. Parmi les méthodes disponibles, citons :

- ◆ public abstract void **close()** throws SQLException
Elle ferme la session, car, ici, le Garbage Collector ne peut agir sur des ressources externes.

- ◆ public abstract void **setAutoCommit(boolean autoCommit)** throws SQLException
Par défaut, toute opération de mise à jour est l'objet d'un commit (c'est-à-dire d'une validation) immédiat. Il est possible d'imposer qu'il n'en est pas ainsi en passant false à cette méthode. La gestion des transactions revient alors au programmeur qui utilisera les méthodes :
 - ◆ public abstract void **commit()** throws SQLException
et
 - ◆ public abstract void **rollback()** throws SQLException

Une des méthodes les plus utiles reste cependant :

```
public abstract Statement createStatement() throws SQLException
```

puisque elle permet de créer l'instruction à exécuter sur la base de données ...

5. L'exécution d'une commande SQL

5.1 Le siège de la commande SQL

En effet, un fois notre connexion attribuée (appelons l'objet *con*), nous allons nous dépêcher de créer une instruction par :

```
Statement instruc = con.createStatement();
```

Un tel objet, implémentant l'interface **Statement**, est utilisé pour transmettre des commandes SQL au SGBD gérant la base de données cible. Il est donc assez logique qu'il soit créé par l'intermédiaire d'un objet connexion qui désigne la base visée.

Les trois méthodes phares de cet interface sont :

- ◆ public abstract ResultSet **executeQuery**(String sql) throws SQLException
- ◆ boolean **execute**(String sql) throws SQLException
- ◆ public abstract int **executeUpdate**(String sql) throws SQLException

dont le rôle est, respectivement, de réaliser une requête de sélection, d'exécuter une commande SQL quelconque (la valeur retournée exprime que son résultat se matérialise par un Resultset ou pas – on peut l'obtenir avec la méthode **getResultSet()**) et d'effectuer une commande de mise à jour sur la bases de données, requête passée dans tous les cas à la méthode sous forme d'une simple chaîne de caractères.

5.2 Une requête de sélection et l'interface ResultSet

D'après ce qui vient d'être dit, une requête de sélection s'effectuera, par exemple, par :

```
ResultSet rs = instruc.executeQuery("select * from stocks");
```

L'objectif est donc bien clairement d'obtenir tous les t-uples de la table "stocks" de la bases de données. Mais où ces t-uples vont-ils être placés ? Un programmeur en SQL-intégré dirait : "dans un *curseur*". Précisément, l'interface **ResultSet** doit conduire à une implémentation d'un tel curseur. Pour rappel, il s'agit en quelque sorte de la table formée des t-uples résultats de la requête SQL, avec cependant la nuance, du moins par défaut, que le déplacement est séquentiel et à sens unique (du début vers la fin).

L'objet implémentant ResultSet gère un pointeur (les anglo-saxons disent, malencontreusement, "*cursor*") gouvernant l'accès aux t-uples résultats. C'est le rôle de la méthode :

```
public abstract boolean next() throws SQLException
```

de passer au t-uple suivant à chacun de ses appels. Le pointeur d'accès est donc, au retour de l'exécution de la requête, placé devant le premier t-uple. Celui-ci est donc le seul à être accessible. Un appel de **next()** chargera le premier t-uple et déplacera le pointeur devant le deuxième t-uple. Et ainsi de suite ... La fin du curseur est détectée par une valeur de retour false de la méthode **next()** : logique !

Comment récupérer les données des divers champs dans des variables Java ? En utilisant les méthodes appelées "getXXX", nom générique des méthodes :

- ◆ String **getString**(String columnName) throws SQLException;
- ◆ boolean **getBoolean**(String columnName) throws SQLException;

- ◆ int **getInt**(String columnName) throws SQLException;
- ◆ long **getLong**(String columnName) throws SQLException;
- ◆ float **getFloat**(String columnName) throws SQLException;
- ◆ double **getDouble**(String columnName) throws SQLException;
- ◆ java.sql.Date **getDate**(String columnName) throws SQLException;
- ◆ java.sql.Time **getTime**(String columnName) throws SQLException;
- ◆ java.sql.Timestamp **getTimestamp**(String columnName) throws SQLException;
- ◆ etc.

Pour notre exemple, on pourra donc écrire :

```
int cpt = 0;
while (rs.next())
{
    if (cpt==0) then System.out.println("Parcours du curseur");
    cpt++;
    String cs = rs.getString("codeSto");
    int x = rs.getInt("x"); int y = rs.getInt("y");
    double q = rs.getDouble("quantite");
    System.out.println(cpt + ". " + cs + " : " + x + "/" + y + " -> " +q);
}
```

Le résultat sera bien conforme à ce que l'on attend :

```
Parcours du curseur
1. 21-123456 : 10/10 -> 650.0
2. 21-123457 : 20/20 -> 780.0
3. 22-123456 : 15/15 -> 56.3
```

Il est aussi possible de récupérer en passant aux méthodes **getXXX**, polymorphes, le numéro de la colonne visée plutôt que son nom :

```
String cs = rs.getString(1);
```

5.3 Les types SQL et les types Java

Lors de l'exécution de toutes la méthodes **getXXX** décrites ci-dessus, le driver JDBC tentera de convertir les données lues dans les tables en un format utilisable pour le type Java attendu. En particulier, on remarquera l'existence d'une classe **Date** au sein du package **java.sql** : il s'agit en fait simplement d'une classe assurant la correspondance entre les dates SQL et les dates instance de la classe Date du package **java.util**; c'est bien cela, c'est une "*wrapper class*" ...

Le tableau suivant met en rapport les instructions **getXXX** permises et recommandées pour les types SQL classiques :

	TINYINT	SMALLINT	INTEGER	BIGINT	REAL	FLOAT	DOUBLE	DECIMAL	NUMERIC	BIT	CHAR	VARCHAR	LONGVARCHAR	BINARY	VARBINARY	LONGVARBINARY	DATE	TIME	TIMESTAMP
getByte	X	x	x	x	x	x	x	x	x	x	x	x	x						
getShort	x	X	x	x	x	x	x	x	x	x	x	x	x						
getInt	x	x	X	x	x	x	x	x	x	x	x	x	x						
getLong	x	x	x	X	x	x	x	x	x	x	x	x	x						
getFloat	x	x	x	x	X	x	x	x	x	x	x	x	x						
getDouble	x	x	x	x	x	X	X	x	x	x	x	x	x						
getBigDecimal	x	x	x	x	x	x	x	X	X	x	x	x	x						
getBoolean	x	x	x	x	x	x	x	x	x	X	x	x	x						
getString	x	x	x	x	x	x	x	x	x	x	X	X	x	x	x	x	x	x	
getBytes														X	X	x			
getDate													x	x	x	X	x		
getTime													x	x	x	X	x		
getTimestamp													x	x	x		x	x	
getAsciiStream													x	x	X	x	x	x	
getUnicodeStream													x	x	X	x	x	x	
getBinaryStream													x	x	X				
getObject	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	

5.4 Une requête de mise à jour

C'est donc la méthode **executeUpdate** qui permet de mettre à jour une table de la base de donnée sur laquelle on possède une connexion. Pour notre exemple :

```
instruc.executeUpdate("update produitsFinis set prixRev =prixRev+10");
```

augmentera les prix de 10 francs pour tous les produits de la table produitsFinis. Les commandes SQL utilisables ici sont toutes celles qui modifient la base de données. Il s'agit donc des commandes DDL : create table, alter table, create index, ... et des commandes DML qui ne sont pas des select : update, insert, delete, ...

6. Le programme complet

Le petit programme suivant résume nos connaissances sur l'utilisation du driver JDBC-ODBC. Pour le plaisir, on y décortique la classe correspondante :

JDBC01.java

```

import java.sql.*;
import java.lang.reflect.*; // uniquement pour l'introspection - pas nécessaire pour jdbc

public class jdbc01
{
    public static void main(String args[]) throws Exception
    {
        System.out.println("Essai de connexion JDBC");
        Class leDriver = Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
        System.out.println("Driver JDBC-OBDC chargé -- Méthodes :");
        // juste pour voir ...
        Method lesMéthodesDuDriver[] = leDriver.getDeclaredMethods();
        for (int i=0; i< lesMéthodesDuDriver.length; i++)
            System.out.println("méthode[" + i + "] = " + lesMéthodesDuDriver[i]);

        // reprenons les choses sérieuses
        Connection con = DriverManager.getConnection("jdbc:odbc:marie","","","");
        System.out.println("Connexion à la BDD marie réalisée");

        Statement instruc = con.createStatement();
        System.out.println("Création d'une instance d'instruction pour cette connexion");

        ResultSet rs = instruc.executeQuery("select * from stocks");
        System.out.println("Instruction SELECT sur stocks envoyée à la BDD marie");
        int cpt = 0;
        while (rs.next())
        {
            if (cpt==0) System.out.println("Parcours du curseur"); cpt++;
            String cs = rs.getString("codeSto");
            System.out.println(" Récupération de codeSto");
            int x = rs.getInt("x"); int y = rs.getInt("y");
            System.out.println(" Récupération de x et y");
            double q = rs.getDouble("quantite");
            System.out.println(" Récupération de quantite");
            System.out.println(cpt + ". " + cs + " : " + x + "/" + y + " -> " +q);
        }

        instruc.executeUpdate("update produitsFinis " +
                            "set prixRev =prixRev+10" );
        System.out.println("Instruction UPDATE sur produitsFinis envoyée à la BDD marie");
        rs = instruc.executeQuery("select * from produitsFinis");
        System.out.println("Instruction SELECT sur produitsFinis envoyée à la BDD marie");
    }
}

```

```
while (rs.next())
{
    String cpf = rs.getString(1);
    double pr = rs.getDouble("prixRev");
    System.out.println(cpf + " : " + pr);
}
}
```

On obtiendra un résultat du type suivant (en focalisant sur l'introspection essentiellement, le contenu de la bases de données étant secondaire ici) :

Essai de connexion JDBC

Driver JDBC-ODBC chargé -- Méthodes :

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[7] = private boolean sun.jdbc.odbc.JdbcOdbcDriver.initialize() throws
java.sql.SQLException**

...

méthode[13] = public java.lang.String

**sun.jdbc.odbc.JdbcOdbcDriver.getConnectionAttributes(java.lang.String,java.lang.String)
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**

Connexion à la BDD marie réalisée

Création d'une instance d'instruction pour cette connexion

Instruction SELECT sur clients envoyée à la BDD marie

Parcours du curseur

...

Remarque

Il est possible d'atteindre une base de données située sur une machine différente :

- ◆ avec ODBC, en créant une source de données distante (bouton "Réseau" lors de la sélection de la source);
- ◆ avec d'autres SGBD, comme déjà signalé, en citant l'adresse de la machine visée avant la base de données proprement dite; par exemple :

Connection *con* = DriverManager.getConnection("jdbc:mysql://192.168.2.1/metal","","");

7. Un curseur plus souple

7.1 Des ResultSet paramétrables

Jusqu'à présent, nos ResultSet ne permettent qu'un parcours séquentiel des tuples sélectionnés; de plus, il n'est pas question de mises à jour de ces tuples en travaillant simplement sur ces ResultSets. Mais, en fait, il est possible d'obtenir des curseurs plus souples, en ce sens que l'on peut demander

- ◆ un type de parcours quelconque (donc, *pas forcément séquentiel*);
- ◆ la possibilité de modifier le tuple traité (donc, le curseur n'est *plus forcément read-only*).

Ceci peut se préciser en utilisant une version polymorphe de la méthode `createStatement()` que possède tout objet implémentant l'interface `Connection` :

```
public Statement createStatement (int resultSetType, int resultSetConcurrency)
    throws SQLException
```

Le premier paramètre détermine le type de déplacement autorisé et peut prendre comme valeur l'une des constantes de classes de `ResultSet` :

<code>public static final int TYPE_FORWARD_ONLY</code>	séquentiel uniquement
<code>public static final int TYPE_SCROLL_SENSITIVE</code>	quelconque – les changements apportés par d'autres opérations sont pris en compte
<code>public static final int TYPE_SCROLL_INSENSITIVE</code>	quelconque – les changements apportés par d'autres opérations ne sont pas pris en compte (type snapshot)

Le deuxième paramètre détermine si le curseur est `read-only` ou `read-write`, au moyen des constantes :

<code>public static final int CONCUR_READ_ONLY</code>	curseur <code>read-only</code>
<code>public static final int CONCUR_UPDATABLE</code>	curseur <code>read-write</code>

Donc, pour pouvoir agir à sa guise sur une version constamment à jour, on créera une connexion à la base de données visée selon :

```
Statement instruc = con.createStatement
    (ResultSet.TYPE_SCROLL_SENSITIVE,
     ResultSet.CONCUR_UPDATABLE);
```

Les méthodes de déplacement dans un `ResultSet` pour lequel de tels déplacements aléatoires sont autorisés sont, en plus de

```
public abstract boolean next() throws SQLException
```

déjà bien connu, les suivantes :

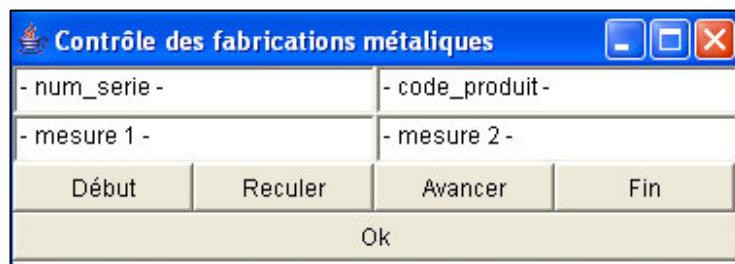
<code>public void beforeFirst() throws SQLException</code>	avant le premier tuple
<code>public boolean first() throws SQLException</code>	sur le premier tuple
<code>public void afterLast() throws SQLException</code>	après le dernier tuple
<code>public boolean last() throws SQLException</code>	sur le dernier tuple
<code>public boolean previous() throws SQLException</code>	reculer d'un tuple
<code>public boolean relative(int rows) throws SQLException</code>	se placer sur le tuple se trouvant à la position précisée

Les méthodes booléennes retournent évidemment false si le déplacement demandé est impossible. Bien sûr, l'exception SQLException est lancée si le déplacement demandé n'est pas permis par le type de connexion utilisé.

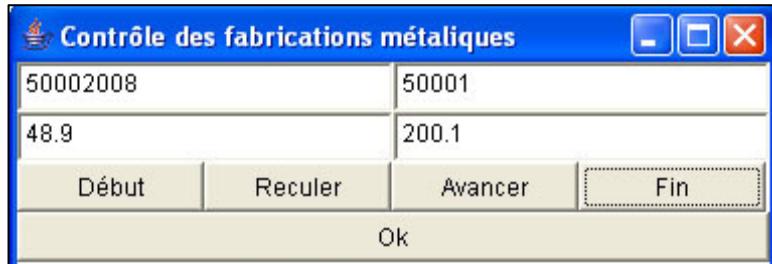
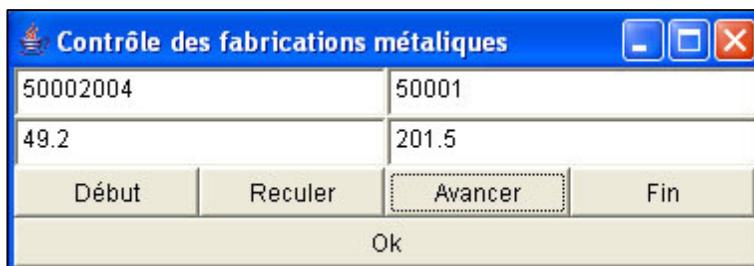
7.2 Un petit viewer de table

Pour illustrer l'utilisation de ces méthodes, nous allons créer un petit viewer de table permettant de parcourir les tuples d'une table "stock" (comportant les champs num_serie, code_produit, mesure_1 et mesure_2) dans une base de données Access "fabricametal" à laquelle nous nous connectés. L'application aura l'aspect suivant :

au démarrage :



au gré des déplacements :



Le code ne présente pas de difficultés particulières – la méthode `getTuple()` ne sert qu'à acquérir les champs du tuple courant et à les placer dans les zones de texte correspondantes du GUI :

FenCurseur (1)

```
import java.sql.*;

public class FenCurseur extends java.awt.Frame
{
    ResultSet rs;

    public FenCurseur()
    {
        initComponents();
        setTitle("Contrôle des fabrications métaliques");

        System.out.println("Essai de connexion JDBC distante");
        try
        {
            Class leDriver = Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            System.out.println("Driver JDBC-OBDC chargé");
        }
        catch (ClassNotFoundException e)
        {
            System.out.println("Driver adéquat non trouvable : " + e.getMessage());
        }

        try
        {
            Connection con = DriverManager.getConnection("jdbc:odbc:fabricametal","","","");
            System.out.println("Connexion à la BDD inpres-metal réalisée");

            Statement instruc = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                ResultSet.CONCUR_UPDATABLE);
            System.out.println("Création d'une instance d'instruction pour cette connexion");

            rs = instruc.executeQuery("select * from stock");
            System.out.println("Instruction SELECT sur stocks envoyée à la BDD marie");
            int cpt = 0;
            while (rs.next())
            {
                if (cpt==0) System.out.println("Parcours du curseur");
                cpt++;
                String ns = rs.getString("num_serie");
                System.out.println(" Récupération de num_serie");
                String cp = rs.getString("code_produit");
                System.out.println(" Récupération de code_produit");
                float m1 = rs.getFloat("mesure_1");
                float m2 = rs.getFloat("mesure_2");
            }
        }
    }
}
```

```

        System.out.println(" Récupération des mesures");
        System.out.println(cpt + ". " + ns + " - " + cp + ":" + m1 + "/" + m2);
    }
}
catch (SQLException e)
{
    System.out.println("Erreur SQL : " + e.getMessage());
}
}

private void initComponents()
{
    panel1 = new java.awt.Panel();
    // instructions pour GUI – passons ...
    ...
}

private void BOkActionPerformed(java.awt.event.ActionEvent evt)
{
    System.exit(0);
}

private void BFinActionPerformed(java.awt.event.ActionEvent evt)
// Bouton Fin
{
    try
    {
        rs.last(); getTuple();
    }
    catch (SQLException e) { System.out.println("Erreur SQL : " + e.getMessage()); }
}

private void BAvancerActionPerformed(java.awt.event.ActionEvent evt) {
// Bouton Acancer
    try
    {
        rs.next(); getTuple();
    }
    catch (SQLException e) { System.out.println("Erreur SQL : " + e.getMessage()); }
}

private void BReculerActionPerformed(java.awt.event.ActionEvent evt)
// Bouton Reculer
{
    try
    {
        rs.previous(); getTuple();
    }
    catch (SQLException e) { System.out.println("Erreur SQL : " + e.getMessage()); }
}

```

```
private void BDebutActionPerformed(java.awt.event.ActionEvent evt)
// Bouton Début
{
    try
    {
        rs.first(); getTuple();
    }
    catch (SQLException e) { System.out.println("Erreur SQL : " + e.getMessage()); }
}

private void exitForm(java.awt.event.WindowEvent evt) { System.exit(0); }

public static void main(String args[])
{
    new FenCurseur().show();
}

private void getTuple () throws SQLException
{
    String ns = rs.getString("num_serie");
    ZTNumSerie.setText(ns);
    String cp = rs.getString("code_produit");
    ZTCodeProduit.setText(cp);
    float m1 = rs.getFloat("mesure_1");
    ZTMesure1.setText(String.valueOf(m1));
    float m2 = rs.getFloat("mesure_2");
    ZTMesure2.setText(String.valueOf(m2));
}

// Variables declaration - do not modify
private java.awt.Button BDebut;
private java.awt.Button BSupprimer;
private java.awt.Button BModifier;
private java.awt.TextField ZTMesure2;
private java.awt.Panel panel2;
private java.awt.TextField ZTCodeProduit;
private java.awt.TextField ZTMesure1;
private java.awt.Button BInserer;
private java.awt.Button BFin;
private java.awt.Panel panel1;
private java.awt.Button BOk;
private java.awt.Panel panel3;
private java.awt.Button BReculer;
private java.awt.Button BAvancer;
private java.awt.TextField ZTNumSerie;
private java.awt.Panel panel4;
// End of variables declaration
}
```

On peut vérifier durant l'exécution que, si on essaie d'avancer quand on est à la fin, ou de reculer si on est au début, on récupère bien une exception :

| Erreur SQL : [Microsoft][Gestionnaire de pilotes ODBC] État de curseur non valide

8. Les mises à jour au travers du curseur

Nous avons dit que le curseur que représente le ResultSet peut être modifiable avec répercussion sur la base de données : c'est ce qui caractérise l'état CONCUR_UPDATABLE de la connexion. Effectivement, on dispose des méthodes nécessaires au sein de ResultSet. Les modifications sont immédiatement prises en compte, du moins si la connexion est en auto-commit (c'est le cas par défaut).

8.1 La modification

Chaque champ peut être mis à jour au moyen des méthodes de ResultSet du type :

```
public void updateString(String columnName, String x) throws SQLException  
public void updateFloat(String columnName, float x) throws SQLException  
...
```

Des méthodes polymorphes utilisant la position du champ visé au lieu de son nom existent également. A ce stade, ce sont seulement les champs du ResultSet qui sont modifiés. Pour que ces changements soient propagés dans la base de données, on appelle simplement la méthode sans paramètre :

```
public void updateRow() throws SQLException
```

8.2 L'insertion

Il faut au préalable appeler la méthode

```
public void moveToInsertRow() throws SQLException
```

qui a comme effet que le curseur est placé sur un tuple particulier, le "*tuple d'insertion*". En clair, il s'agit d'un buffer structuré comme un tuple et qui est la matérialisation du futur nouveau tuple. On peut dès lors placer des valeurs dans les champs au moyen des méthodes updateXXX() évoquées au point précédent. L'insertion effective se fait au moyen de la méthode :

```
public void insertRow() throws SQLException
```

8.3 La suppression

L'opération est fort simple, puisqu'il suffit de se positionner sur le tuple victime et d'appeler la méthode :

```
public void deleteRow() throws SQLException
```

8.4 Un viewer de table plus complet

Nous pouvons à présent compléter notre application précédente en y ajoutant les opérations classiques de mise à jour :



L'application a gagné trois méthodes de plus :

FenCurseur (2)

```
import java.sql.*;

public class FenCurseur extends java.awt.Frame
{
    ResultSet rs;

    public FenCurseur()
    {
        initComponents(); setTitle("Contrôle des fabrications métalliques");
        System.out.println("Essai de connexion JDBC distante");
        try
        {
            Class leDriver = Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            System.out.println("Driver JDBC-OBDC chargé");
        }
        catch (ClassNotFoundException e)
        { System.out.println("Driver adéquat non trouvable : " + e.getMessage()); }

        try
        {
            Connection con = DriverManager.getConnection("jdbc:odbc:fabricametal","","");
            System.out.println("Connexion à la BDD inpres-metal réalisée");
            Statement instruc = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                ResultSet.CONCUR_UPDATABLE);
            System.out.println("Création d'une instance d'instruction pour cette connexion");
            ...
        }
        catch (SQLException e) { System.out.println("Erreur SQL : " + e.getMessage()); }
    }
}
```

```

...
private void BModifierActionPerformed (java.awt.event.ActionEvent evt)
{
    try
    {
        rs.updateString("num_serie", ZTNumSerie.getText());
        rs.updateString("code_produit", ZTCodeProduit.getText());
        rs.updateFloat("mesure_1", Float.parseFloat(ZTMesure1.getText()));
        rs.updateFloat("mesure_2", Float.parseFloat(ZTMesure2.getText()));

        rs.updateRow();
    }
    catch (SQLException e) { System.out.println("Erreur SQL : " + e.getMessage()); }
}

private void BInsererActionPerformed (java.awt.event.ActionEvent evt)
{
    try
    {
        rs.moveToInsertRow();
        rs.updateString("num_serie", ZTNumSerie.getText());
        rs.updateString("code_produit", ZTCodeProduit.getText());
        rs.updateFloat("mesure_1", Float.parseFloat(ZTMesure1.getText()));
        rs.updateFloat("mesure_2", Float.parseFloat(ZTMesure2.getText()));
        rs.insertRow();
    }
    catch (SQLException e) { System.out.println("Erreur SQL : " + e.getMessage()); }
}

private void BSupprimerActionPerformed (java.awt.event.ActionEvent evt)
{
    try
    {
        rs.deleteRow();
    }
    catch (SQLException e) { System.out.println("Erreur SQL : " + e.getMessage()); }
}

...

public static void main(String args[])
{
    new FenCurseur().show();
}

...
// Variables declaration - do not modify
private java.awt.Button BSupprimer;
private java.awt.Button BModifier;
private java.awt.Button BInserer;
...
}

```

9. Un peu de SQL dynamique

Evidemment, les amateurs de bases de données auront remarqué que les requêtes envisagées ci-dessus sont statiques, c'est-à-dire écrites une bonne fois pour toutes dans le code de l'application. En pratique, il est cependant parfois nécessaire de laisser certaines parties de la requête en variables, afin de lui donner plus de généralité. On aura clairement reconnu ici les objectifs du SQL dynamique.

Dans cet esprit, Java propose un interface **PreparedStatement** dérivé de Statement. Son rôle est analogue à celui de son père, si ce n'est que la requête qui lui donnera naissance comporte des paramètres indiqués par "?", par exemple :

```
"update produitsFinis set prixRev = prixRev * ? "+  
" where codeSto = ?"
```

Les méthodes du type

```
public abstract void setXXX (int positionParametre, XXX valeurParametre)  
throws SQLException
```

permettent alors de fixer la valeur des paramètres "?" de la requête selon leur position. Le type de ces paramètres peut être quelconque, puisque les méthodes setXXX sont très nombreuses :

- ◆ setBoolean(int, boolean)
- ◆ setByte(int, byte)
- ◆ setBytes(int, byte[])
- ◆ setDate(int, Date)
- ◆ setDouble(int, double)
- ◆ setFloat(int, float)
- ◆ setInt(int, int)
- ◆ setLong(int, long)
- ◆ setNull(int, int)
- ◆ ...

Une fois la valeur des paramètres fixées, il reste à exécuter la requête comme d'habitude par executeQuery() ou executeUpdate().

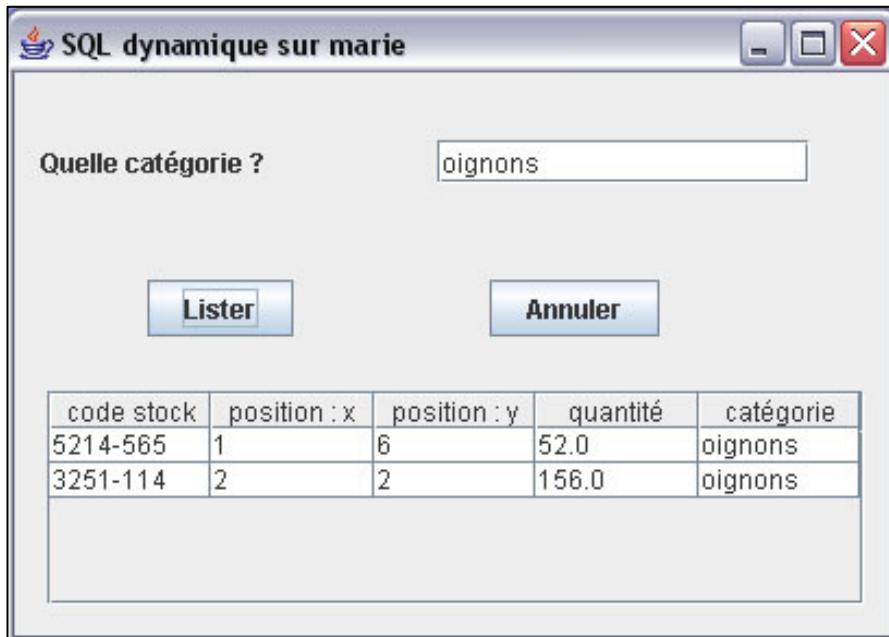
L'objet implémentant l'interface PreparedStatement sera obtenu par l'intermédiaire de l'objet implémentant **Connection**, lequel appellera sa méthode :

```
public abstract PreparedStatement prepareStatement(String sql) throws SQLException;
```

On peut ainsi compléter le programme précédent, par exemple, des instructions suivantes :

```
PreparedStatement pStmt =  
    con.prepareStatement("produitsFinis set prixRev = prixRev * ? "+  
                        " where codeSto = ?");  
pStmt.setInt(1, 20000);  
pStmt.setString(2, "JAM007");  
pStmt.executeUpdate();
```

mais, évidemment, ceci présente surtout de l'intérêt si la valeur de prixRev et celle de codeStock sont **obtenues dynamiquement** au moment de l'exécution, par exemple au moyen d'un GUI. C'est ce que propose l'application dont l'aspect est le suivant :



et dont le code principal (développé sous Netbeans) est (remarquer l'insertion des informations des tuples retenus dans la JTable) :

FenAppJdbcDyna.java

```
/*
 * FenAppJdbcDyna.java
 */

package jdbcdyna;

/**
 * @author vilvens
 */

import java.sql.*;
import java.util.*;
import javax.swing.*;
import javax.swing.table.*;

public class FenAppJdbcDyna extends javax.swing.JFrame
{
    public FenAppJdbcDyna() { initComponents(); }

    private void initComponents()
    {
        BLister = new javax.swing.JButton();
        TableauTuples = new javax.swing.JTable();
    }
}
```

```

TFCategorie = new javax.swing.JTextField();
...
}

private void BListerActionPerformed(java.awt.event.ActionEvent evt)
{
    String categ = TFCategorie.getText();
    System.out.println("Catégorie choisie : " + categ);
    try
    {
        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
        System.out.println("Driver JDBC-OBDC chargé");
        Connection con = DriverManager.getConnection("jdbc:odbc:marie",
            "genius","grosZZ");
        System.out.println("Connexion à la BDD marie-inpres réalisée");

PreparedStatement pStmt =
    con.prepareStatement("select * from stocks where categorie = ?");
    pStmt.setString(1, categ);

ResultSet rs = pStmt.executeQuery();
    System.out.println("Instruction SELECT sur stocks envoyée à la BDD marie");
    int cpt = 0;
    if (!rs.next())
    {
        System.out.println("Aucun tuple trouvé :-( !");
        return;
    }
    do
    {
        Vector ligne = new Vector();
        if (cpt==0) System.out.println("Parcours du curseur"); cpt++;
        String cs = rs.getString("codeSto"); ligne.add(cs);
        int x = rs.getInt("x"); ligne.add(String.valueOf(x));
        int y = rs.getInt("y");ligne.add(String.valueOf(y));
        double q = rs.getDouble("quantite"); ligne.add(String.valueOf(q));
        String c = rs.getString(5); ligne.add(c);
        System.out.println(cpt + ". " + cs + ":" + x + "/" + y + " -> " +q + "["+c+"]");

DefaultTableModel dtm = (DefaultTableModel)TableauTuples.getModel();
    dtm.insertRow(dtm.getRowCount(),ligne);
}
while (rs.next());
}
catch (SQLException ex)
{ System.out.println("Erreur SQL : " + ex.getMessage()); }
catch (ClassNotFoundException ex)
{ System.out.println("Driver adéquat non trouvable : " + ex.getMessage()); }
}

```

```
public static void main(String args[]) {
    java.awt.EventQueue.invokeLater(new Runnable() {
        public void run() {
            new FenAppJdbcDyna().setVisible(true);
        }
    });
}

private javax.swing.JButton BLISTER;
private javax.swing.JTextField TFCategorie;
private javax.swing.JTable TableauTuples;
...
}
```

Remarque

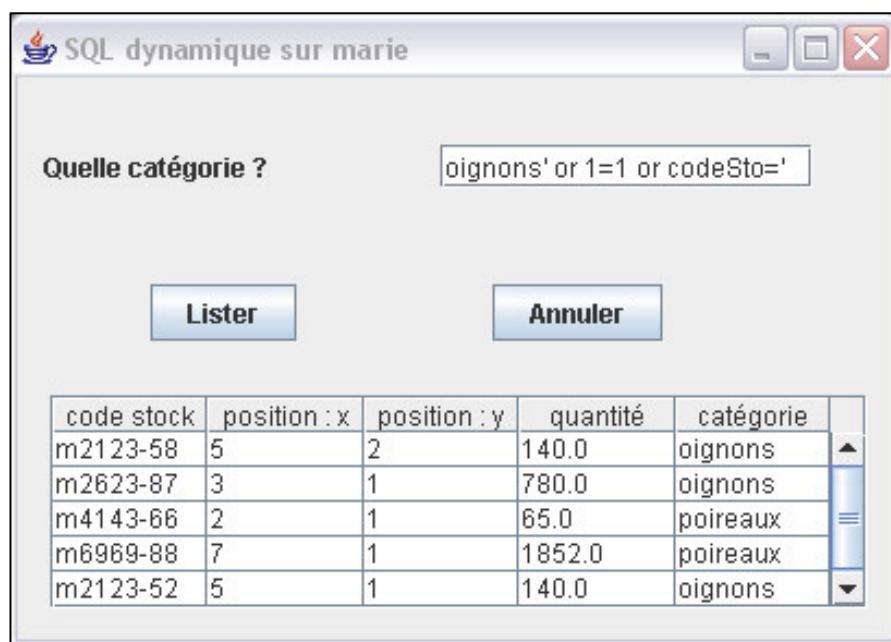
On pourrait penser obtenir un résultat analogue avec un simple appel de la méthode `executeQuery()` à qui l'on passerait une chaîne de caractères représentant la requête et qui aurait été créée dynamiquement, donc quelque chose du genre (pour notre exemple précédent) :

```
Statement instruc = con.createStatement();
String requete = "select * from stocks where categorie = '" + categ + "';";
ResultSet rs = instruc.executeQuery(requete);
```

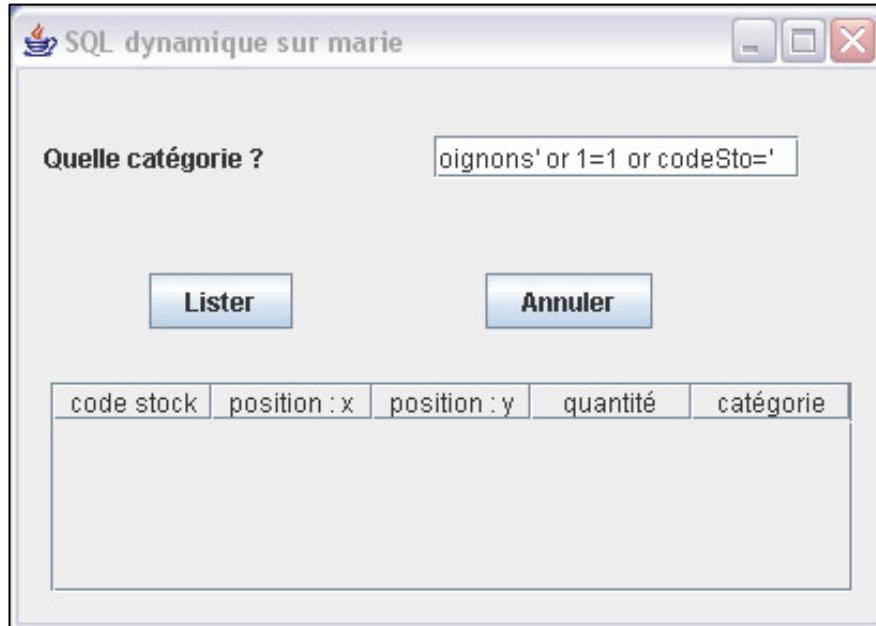
Mais on offre alors le flanc à l'attaque connue sous le nom de "**SQL injection**" : il s'agit d'introduire une requête malicieuse permettant d'obtenir bien plus que ce que l'on devrait normalement obtenir. Ainsi, dans le cas de notre exemple, l'entrée pour la catégorie de :

oignons' or 1=1 or codeSto='

donne la liste complète ☺ !



On imagine le problème s' il n'était pas question d'oignons ou de poireaux, mais de mot de passe ... L'utilisation d'un PreparedStatement à la place d'un simple Statement rend ce genre d'attaque bien plus irréalisable (ne soyons pas trop catégorique ;-)) : ainsi, la version Sql-dynamique :



donne sur la console :

```
Catégorie choisie : oignons' or 1=1 or codeSto=''
Aucun tuple trouvé :-( !
```

10. L'utilisation des dates et heures

Tout qui s'y est déjà frotté le sait : l'utilisation des dates et heures n'est pas toujours chose facile dans le contexte des bases de données. Nous avons déjà signalé l'existence des méthodes :

- ◆ `java.sql.Date getDate(String columnName) throws SQLException;`
- ◆ `java.sql.Time getTime(String columnName) throws SQLException;`
- ◆ `java.sql.Timestamp getTimestamp(String columnName) throws SQLException;`

Histoire d'insister, le petit programme suivant illustre leur utilisation, en utilisant une base de données "meteo" contenant dans une table "alarmes" des alarmes pour un lieu donné à une heure donnée :

- ◆ la première partie lit, entre autres, un groupe heure et en extrait la seule heure;
- ◆ la seconde partie met à jour une heure dans la base en utilisant une instruction de sql dynamique.

Jdbc02.java

```

import java.sql.*;
import java.util.*;

public class jdbc02
{
    public static void main(String args[])
    {
        try
        {
            System.out.println("Essai de connexion JDBC");
            Class leDriver = Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            System.out.println("Driver JDBC-OBDC chargé");
            Connection con = DriverManager.getConnection("jdbc:odbc:meteo");
            System.out.println("Connexion à la BDD marie réalisée");
            Statement instruc = con.createStatement();
            System.out.println("Création d'une instance d'instruction pour cette connexion");

            // 1. Lectures avec une heure
            ResultSet rs = instruc.executeQuery("select * from alarmes");
            System.out.println("Instruction SELECT sur alarmes envoyée à la BDD meteo");
            int cpt = 0;
            while (rs.next())
            {
                if (cpt==0) then System.out.println("Parcours du curseur");
                cpt++;
                String l = rs.getString("lieu");
                int x = rs.getInt("numAlarme");
                Time t = rs.getTime("heure");
                System.out.println(x + ". " + l + " : " + t);

                GregorianCalendar gc = new GregorianCalendar();
                gc.setTime(t);
                int heure = gc.get(Calendar.HOUR_OF_DAY);
                System.out.println("    " + heure );
            }

            // 2. Mise à jour d'une heure
            Time t = new Time(23,55,12);
            PreparedStatement ps =
                con.prepareStatement("update alarmes set heure = ? where numAlarme = 2" );
            ps.setTime(1, t);
            ps.executeUpdate();
            System.out.println("Instruction UPDATE sur alarmes envoyée à la BDD meteo");

            rs = instruc.executeQuery("select * from alarmes");
            System.out.println("Instruction SELECT sur alarmes envoyée à la BDD meteo (2)");
            while (rs.next())
            {

```

```
if (cpt==0) then System.out.println("Parcours du curseur");
cpt++;
String l = rs.getString("lieu");
int x = rs.getInt("numAlarme");
t = rs.getTime("heure");
System.out.println(x + ". " + l + " : " + t);
GregorianCalendar gc = new GregorianCalendar();
gc.setTime(t);
int heure = gc.get(Calendar.HOUR_OF_DAY);
System.out.println("    " + heure );
}
}
catch (ClassNotFoundException e)
{
    System.out.println(":-) Erreur JDBC : " + e.getMessage());
}
catch (SQLException e)
{
    System.out.println(":-) Erreur SQL : " + e.getMessage());
}
}
}
```

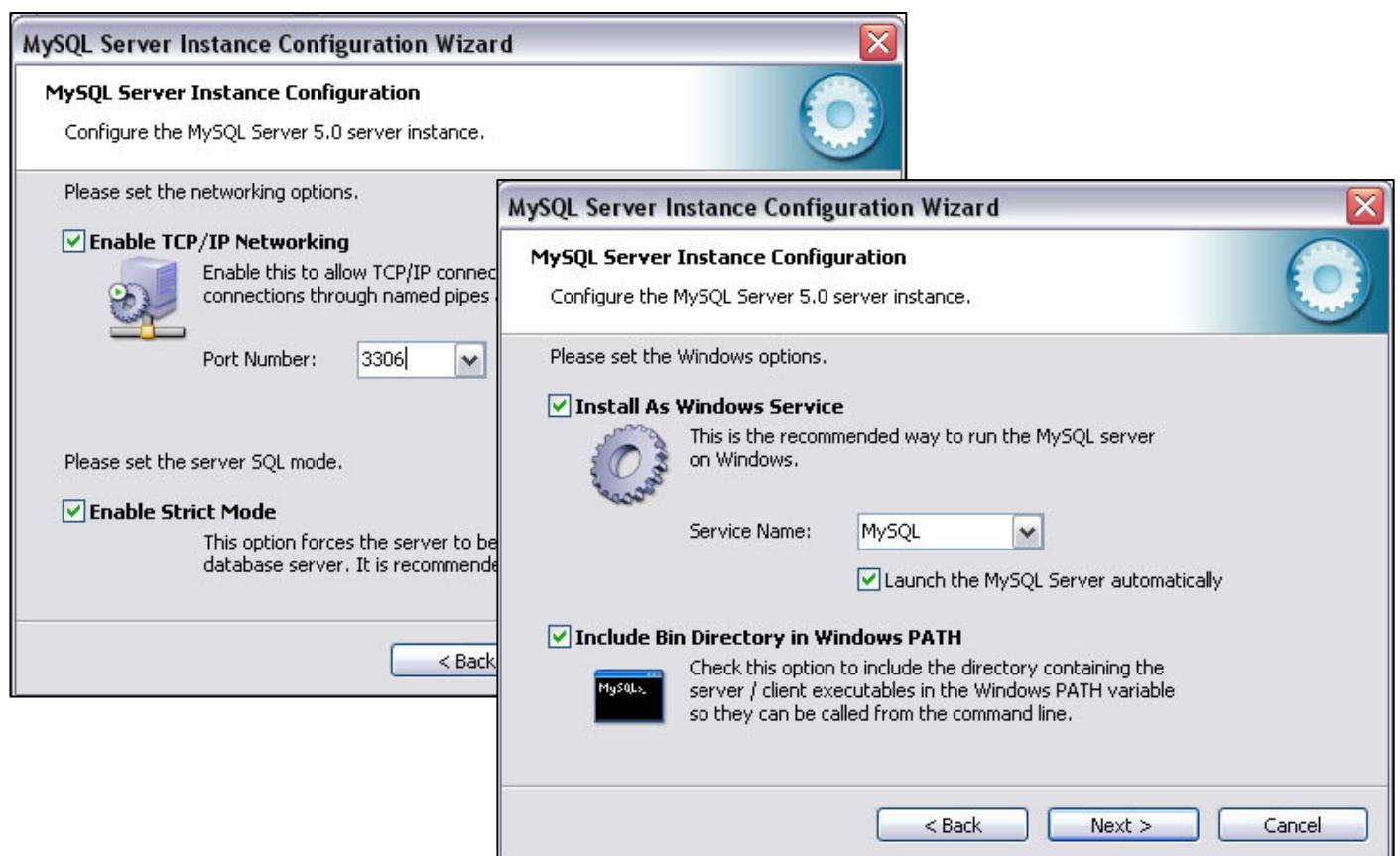
Résultat sur la console :

```
Essai de connexion JDBC
Driver JDBC-OBDC chargé
Connexion à la BDD marie réalisée
Création d'une instance d'instruction pour cette connexion
Instruction SELECT sur alarmes envoyée à la BDD meteo
Parcours du curseur
1. Oupeye : 12:23:01
    12
2. Seraing : 10:51:36
    10
3. Liège : 08:23:41
    8
Instruction UPDATE sur alarmes envoyée à la BDD meteo
Instruction SELECT sur stocks envoyée à la BDD meteo
1. Oupeye : 12:23:01
    12
2. Seraing : 23:55:12
    23
3. Liège : 08:23:41
    8
```

11. Un autre exemple JDBC avec MySql

11.1 Le serveur MySql

On peut télécharger sur www.mysql.com un serveur mysql pour Windows – ici, on a téléchargé la version 5. L'installation permet de préciser des paramètres importants :





On a pu voir que le serveur MySQL a donc été lancé comme un service qui restera donc disponible durant l'ensemble de la session. Le choix dans "démarrer" de la ligne de commande de MySQL donne dans une fenêtre DOS :

```
Enter password: *****
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 5
Server version: 5.0.41-community-nt MySQL Community Edition (GPL)
```

Type 'help;' or 'h' for help. Type 'c' to clear the buffer.

```
mysql> show databases;
+-----+
| Database      |
+-----+
| information_schema |
| mymarieinpres   |
| mysql          |
| test           |
+-----+
4 rows in set (0.00 sec)
```

La base de données mysql est en fait la base d'administration. On peut s'en faire une idée en la consultant :

```
mysql> use mysql
Database changed
mysql> show tables;
+-----+
| Tables_in_mysql |
+-----+
```

```
| columns_priv
| db
| func
| help_category
| help_keyword
| help_relation
| help_topic
| host
| proc
| procs_priv
| tables_priv
| time_zone
| time_zone_leap_second
| time_zone_name
| time_zone_transition
| time_zone_transition_type
| user
+
+-----+
17 rows in set (0.02 sec)
```

Deux tables méritent d'être signalées :

- ◆ user qui contient le profil des utilisateurs;
 - ◆ db qui contient évidemment les caractéristiques de bases de données existant sur le serveur :

```
mysql> select * from db;
+-----+-----+-----+-----+-----+
|-----+-----+-----+-----+-----+
|-----+-----+-----+-----+-----+
|-----+-----+-----+-----+-----+
|-----+-----+-----+-----+-----+
| Host | Db      | User   | Select_priv | Insert_priv | Update_priv | Del
ete_priv | Create_priv | Drop_priv | Grant_priv | References_priv | Index_priv |
Alter_priv | Create_tmp_table_priv | Lock_tables_priv | Create_view_priv | Show
_view_priv | Create_routine_priv | Alter_routine_priv | Execute_priv |
+-----+-----+-----+-----+-----+
|-----+-----+-----+-----+-----+
|-----+-----+-----+-----+-----+
|-----+-----+-----+-----+-----+
|-----+-----+-----+-----+-----+
| %   | mymarieinpres | vilvens | Y          | Y          | Y          | Y
| Y    | Y          | N          | Y          | Y          | Y          |
| Y    | Y          | Y          | Y          | Y          | Y          |
| Y    | Y          | Y          | Y          |           |
+-----+-----+-----+-----+-----+
|-----+-----+-----+-----+-----+
|-----+-----+-----+-----+-----+
|-----+-----+-----+-----+-----+
|-----+-----+-----+-----+-----+
1 row in set (0.02 sec)
```

Un utilisateur préalablement créé (disons "vilvens") se connectera à une base de données également préalablement créée (disons "myMarieInpres" – un clone de la base Access utilisée ci-dessus) en utilisant une fenêtre DOS :

```
C:\Documents and Settings\Vilvens>mysql -u vilvens -p
Enter password: *****
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 12
Server version: 5.0.41-community-nt MySQL Community Edition (GPL)
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> use myMarieInpres
Database changed
mysql> select * from stocks;
+-----+-----+-----+-----+
| codeSto | x   | y   | quantite | categorie |
+-----+-----+-----+-----+
| m2123-58 | 5 | 2 | 140 | oignons |
| m2623-87 | 3 | 1 | 780 | oignons |
| m4143-66 | 2 | 1 | 65 | poireaux |
| m6969-88 | 7 | 1 | 1852 | poireaux |
| m2123-52 | 5 | 1 | 140 | oignons |
+-----+-----+-----+-----+
5 rows in set (0.00 sec)

mysql>
```

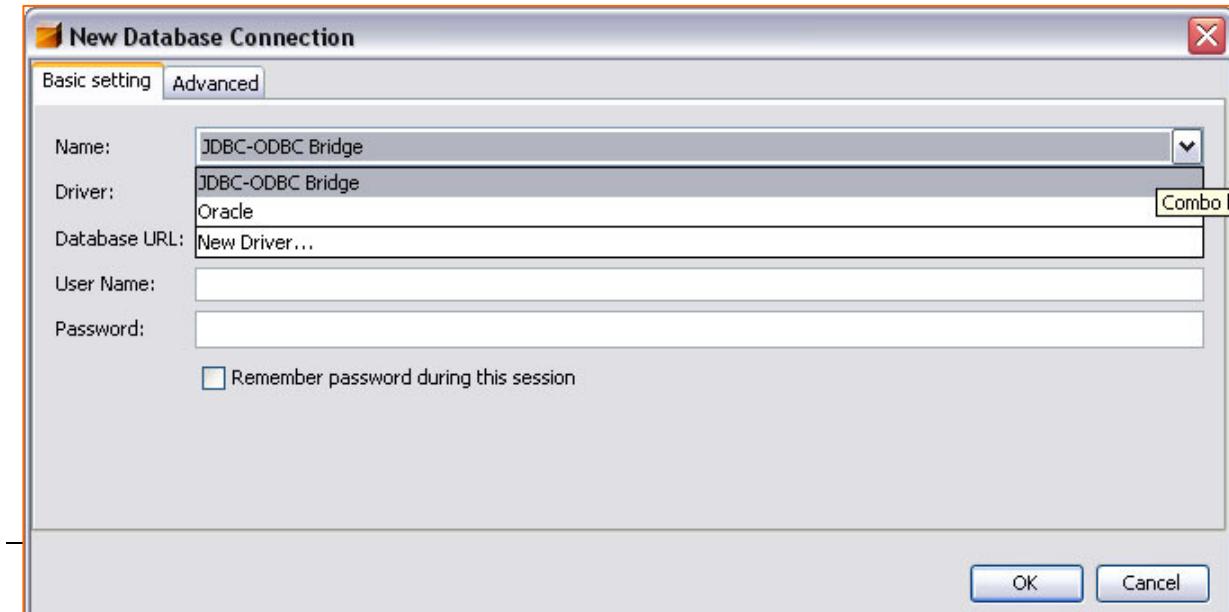


Avec moi, c'est plus facile;-)

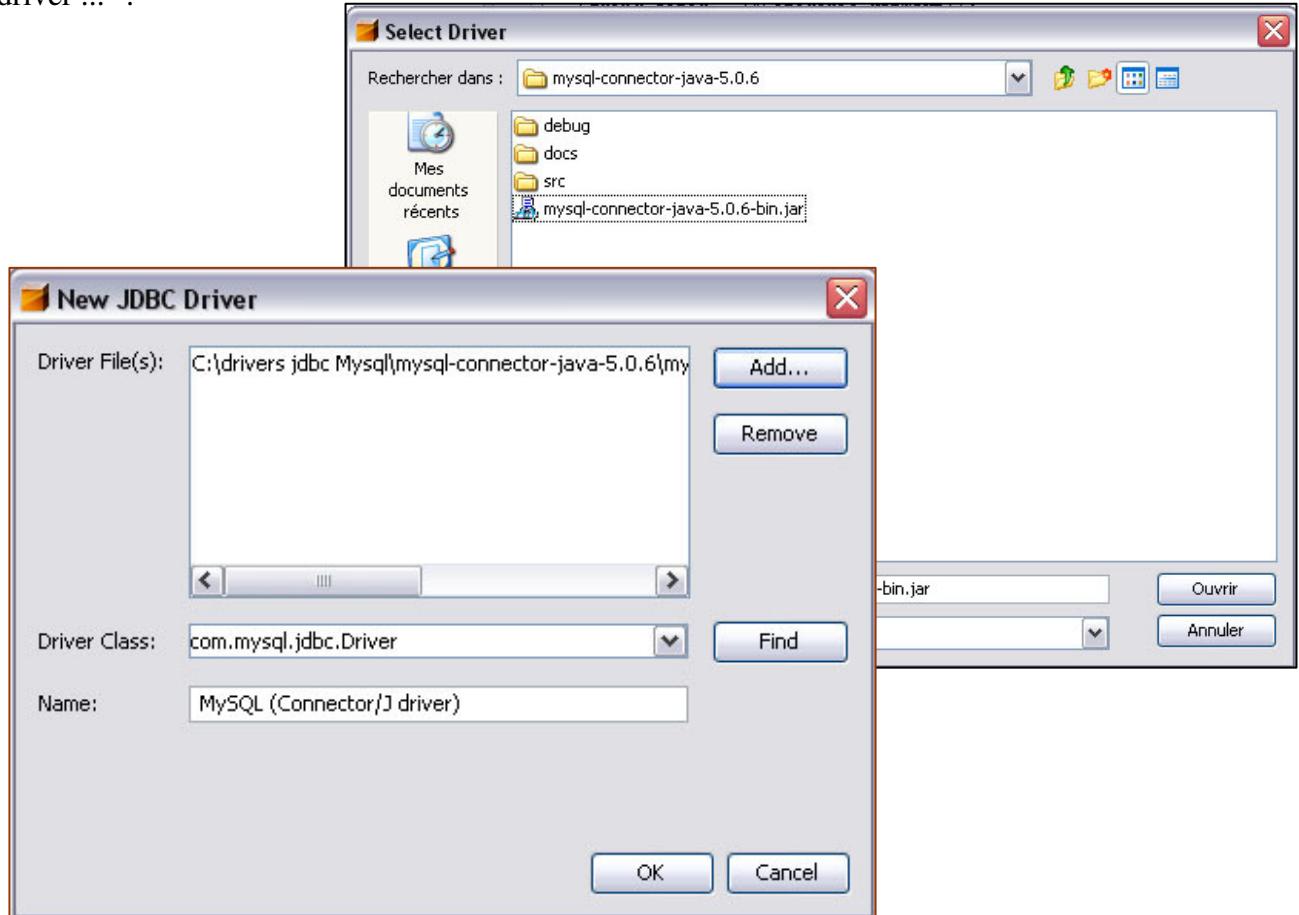
On pourra vérifier que la base de données mymarieinpres se trouve dans le répertoire C:\Program Files\MySQL\MySQL Server 5.0\data, à côté de la base système mysql.

11.2 Utilisation des bases de données dans Netbeans

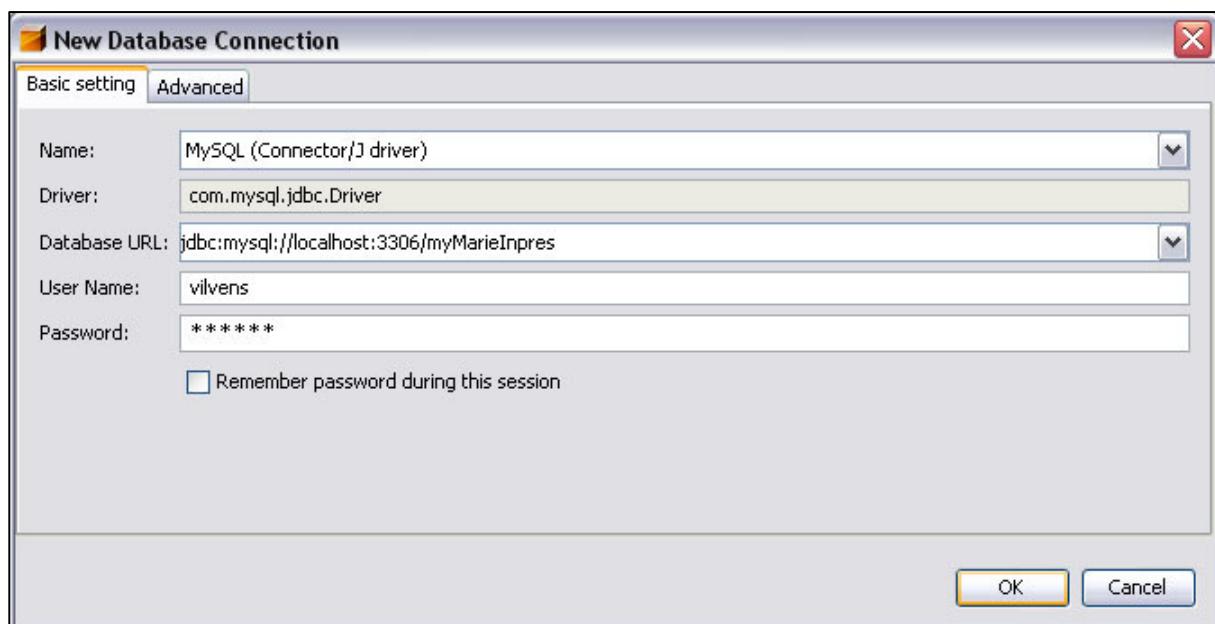
Après cette incursion dans le monde des bases de données, revenons-en à nos développements Java. Dans NetBeans 5.5 ou 6.*, il est possible de se connecter de manière permanente à une base de données : il suffit, dans l'onglet Runtime ou Services, de réaliser un clic droit sur le nœud Databases et choisir "New Connection" :



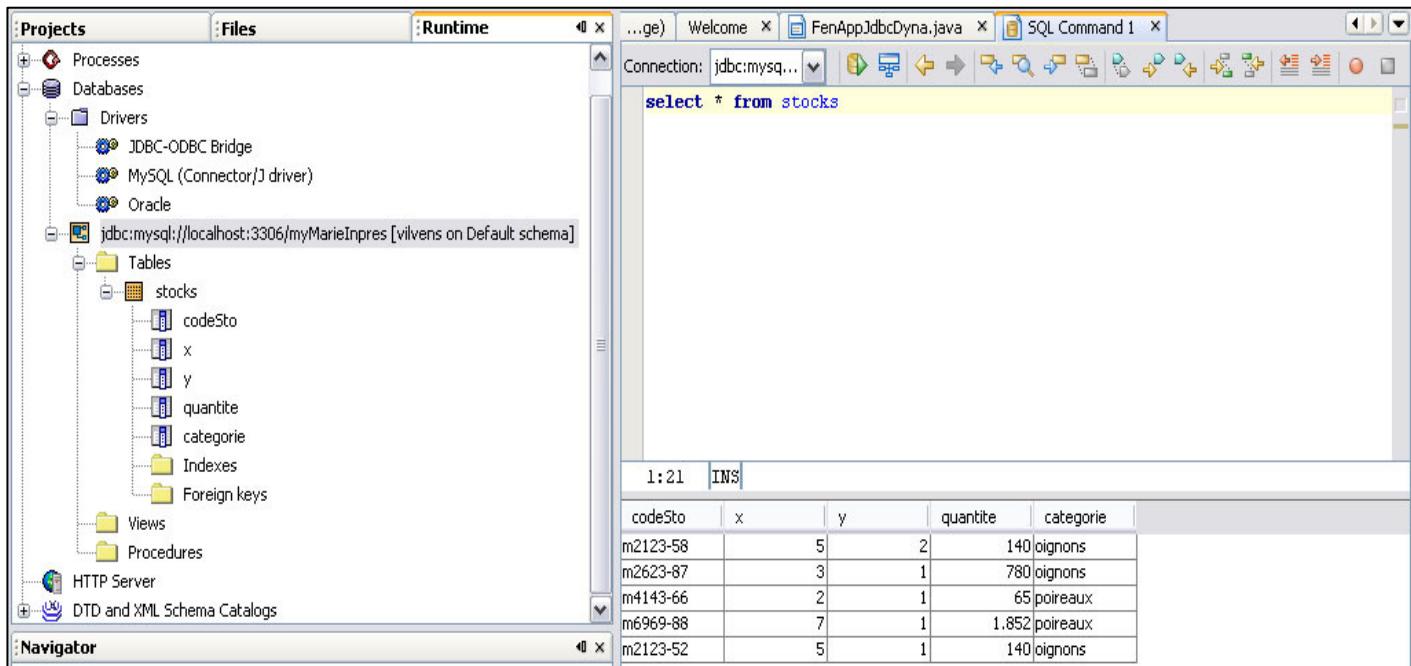
Si nous visons notre base de données MySQL, il nous faudra tout d'abord charger le driver jdbc correspondant. Un clic droit sur le nœud Drivers nous permet de choisir "New driver ..." :



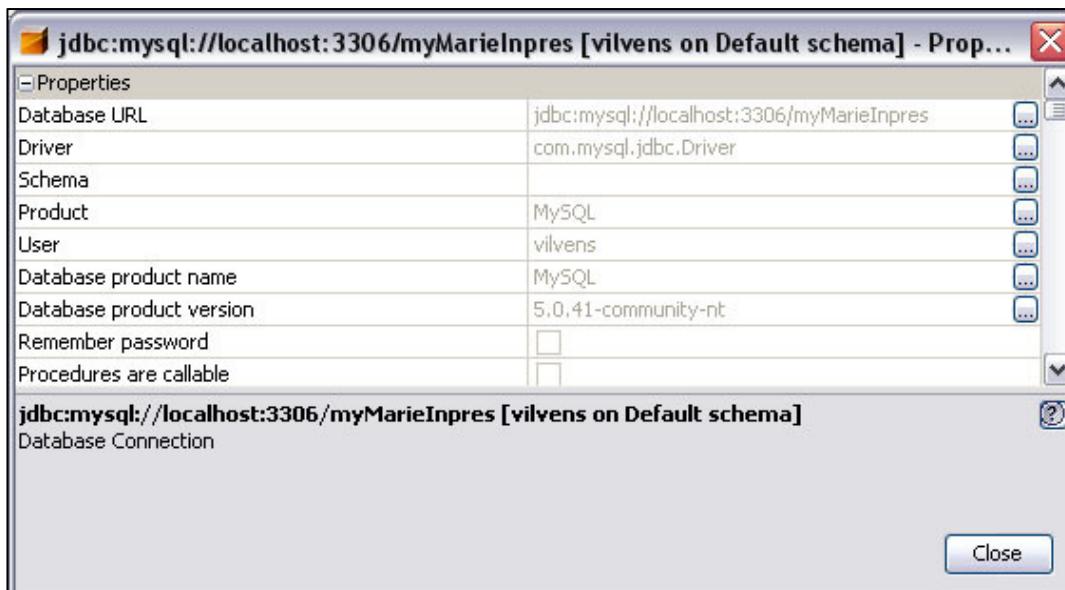
Restera alors à choisir la base visée :



On peut alors constater que l'on peut visualiser la base choisie en détails :



Bien sûr, si l'on accède les propriétés :



11.3 La portabilité des applications par rapport aux SGBDs

Nous l'avons dit, cette base MySQL est un clone de la base Access marie-inpres. Le programme ci-dessus qui accédait cette dernière peut donc être repris tel quel moyennant les deux modifications évidentes :

```
Class.forName("com.mysql.jdbc.Driver");
System.out.println("Driver MySql chargé");
Connection con = DriverManager.getConnection
    ("jdbc:mysql://localhost:3306/myMarieInpres","vilvens","grosZZ");
```

On obtient immédiatement (le choix "poireaux" a déjà été validé) :



12. Les métadonnées

12.1 Obtenir des informations sur une table

Les metadata, c'est-à-dire les données sur la base de données et ses tables, sont accessible en utilisant la méthode de ResultSet :

```
public abstract ResultSetMetaData getMetaData() throws SQLException
```

Ce que l'on obtient est un objet qui implémente l'interface **ResultSetMetaData** qui contient toutes les méthodes utiles pour obtenir les informations souhaitées – par exemple (les significations semblent claires) :

```
public abstract int getColumnCount() throws SQLException
```

et pour chaque colonne :

```
String getTableName(int column) throws SQLException
String getColumnName(int column) throws SQLException
String getColumnTypeName(int column) throws SQLException
String getColumnClassName(int column) throws SQLException
```

En reprenant un programme précédent décliné pour la base de données MySQL :

```
JDBCSimple.java (avec metadata)
/*
 * JDBCSimple.java
 */
package jdbc;

/**
```

```

* @author vilvens
*/
import java.sql.*;

public class JDBCSSimple
{
    public static void main(String args[]) throws Exception
    {
        System.out.println("Essai de connexion JDBC");
        Class leDriver = Class.forName("com.mysql.jdbc.Driver");
        System.out.println("Driver JDBC-MySQL chargé");
        Connection con =
        DriverManager.getConnection("jdbc:mysql://localhost:3306/myMarieInpres",
            "vilvens", "grosZZ");
        System.out.println("Connexion à la BDD marie-inpres réalisée");

        Statement instruc = con.createStatement();
        System.out.println("Création d'une instance d'instruction pour cette connexion");

        ResultSet rs = instruc.executeQuery("select * from stocks");
        System.out.println("Instruction SELECT sur stocks envoyée à la BDD marie");
        int cpt = 0;
        while (rs.next())
        {
            if (cpt==0) System.out.println("Parcours du curseur"); cpt++;
            String cs = rs.getString("codeSto");
            int x = rs.getInt("x"); int y = rs.getInt("y");
            double q = rs.getDouble("quantite");
            String categ = rs.getString("categorie");
            System.out.println(cpt + ". " + cs + " : " + x + "/" + y + " -> " +q);
        }

        /** Méta-données
        ResultSetMetaData rsmd = rs.getMetaData();

        int nbcoll = rsmd.getColumnCount();
        System.out.println("Nom de la table : " + rsmd.getTableName(1));
        System.out.println("Nombre de colonnes = " + nbcoll);
        for (int i=1; i<= nbcoll; i++)
        {
            System.out.print("Nom de colonne : " + rsmd.getColumnName(i));
            System.out.print(" - Type de colonne : " + rsmd.getColumnTypeName(i));
            System.out.println(" - Nom de la classe Java équivalente : " +
                rsmd.getColumnClassName(i));
        }
    }
}

```

on obtient comme résultat :

Essai de connexion JDBC

Driver JDBC-MySQL chargé

Connexion à la BDD marie-inpres réalisée

Création d'une instance d'instruction pour cette connexion

Instruction SELECT sur stocks envoyée à la BDD marie

Parcours du curseur

1. m2123-58 : 5/2 -> 140.0

2. m2623-87 : 3/1 -> 780.0

3. m4143-66 : 2/1 -> 65.0

4. m6969-88 : 7/1 -> 1852.0

5. m2123-52 : 5/1 -> 140.0

Nom de la table : stocks

Nombre de colonnes = 5

Nom de colonne : *codeSto* - Type de colonne : **VARCHAR** - Nom de la classe Java

équivalente : java.lang.String

Nom de colonne : *x* - Type de colonne : **INTEGER** - Nom de la classe Java équivalente : java.lang.Integer

Nom de colonne : *y* - Type de colonne : INTEGER - Nom de la classe Java équivalente : java.lang.Integer

Nom de colonne : *quantite* - Type de colonne : **DOUBLE** - Nom de la classe Java équivalente : java.lang.Double

Nom de colonne : *categorie* - Type de colonne : VARCHAR - Nom de la classe Java équivalente : java.lang.String

Il est donc possible de faire de l'introspection sur les colonnes d'une table ;-)

12.2 Une application : les requêtes scalaires

A titre d'illustration de l'usage des métadonnées, intéressons-nous à la requête :

```
select count(*) from alarmes
```

pour la base "meteo" déjà évoquée. Ce type de requête n'est pas fort bien documentée.

Explorons-la donc pour voir ce qui se passe si nous l'exécutons au travers de JDBC, au moyen de la méthode execute() qui nous permettra de savoir si un Resultset en a résulté. Nous pouvons en fait constater que le résultat du count(*) est effectivement placé dans un Resultset, comportant évidemment un seul tuple à un seul champ :

Jdbc02.java (suite)

```
...
    boolean retour = instruc.execute("select count(*) from alarmes");
    System.out.println("Instruction SELECT COUNT(*) sur alarmes envoyée à la BDD
meteo");
    System.out.println("retour = " + retour);
    ResultSet rsc = instruc.getResultSet();
    if (rsc==null) System.out.println("Pas de resultset");
    else System.out.println("Resultset récupéré");

    ResultSetMetaData rsmd = rsc.getMetaData();
    int nbreCol = rsmd.getColumnCount();
```

```

System.out.println("Nombre de champs dans le resultset = " + nbreCol);
String nomCol = rsmd.getColumnName(1);
System.out.println("Nom du champ dans le resultset = " + nomCol);
int typeCol = rsmd.getColumnType(1);
System.out.println("Type du champ dans le resultset = " + typeCol);
String nomTypeCol = rsmd.getColumnTypeName(1);
System.out.println("Nom du type du champ dans le resultset = " + nomTypeCol);

int index = rsc.findColumn("Expr1000");
System.out.println("index = " + index); // vérification
if (rsc.next()) // ne pas oublier : le pointeur est devant le premier record du curseur
{
    short nbre = rsc.getShort(1); // ou rsc.getShort("Expr1000");
    System.out.println("Nombre de tuples = " + nbre);
}
...

```

Instruction SELECT COUNT(*) sur alarmes envoyée à la BDD meteo

retour = true

Resultset récupéré

Nombre de champs dans le resultset = 1

Nom du champ dans le resultset = Expr1000

Type du champ dans le resultset = 4

Nom du type du champ dans le resultset = INTEGER

index = 1

Nombre de tuples = 3

Donc, en résumé, la marche à suivre pour une telle requête scalaire se résume à ceci :

```

ResultSet rsco = instruc.executeQuery("select count(*) from alarmes");
System.out.println("Instruction SELECT COUNT(*) - bis -sur alarmes envoyée à la
BDD meteo");
if (rsco.next())
{
    short nbre = rsc.getShort(1);
    System.out.println("Nombre de tuples = " + nbre);
}

```

13. Erreurs, transactions et procédures stockées

JDBC offre encore d'autres possibilités. Citons sommairement :

- ◆ **le traitement des erreurs** : l'exception SQLException comporte, outre un message imprimable (comme toutes les exceptions), la valeur de la variable **SQLSTATE** et celle du code d'erreur propriétaire du fabricant du SGBD utilisé. On peut obtenir ces valeurs par les méthodes :

```

public String getSQLState()
public int getErrorCode()

```

Un exemple classique de traitement d'erreur SQL est alors :

```
catch (SQLException e)
{
    sortie.println("Erreur JDBC-OBDC : " + e.getMessage() + "(" + e.getSQLState() + ")");
}
```

♦ **les transactions** : pour rappel (voir paragraphe 5), une connexion à une base de données au moyen de JDBC est en mode auto-commit par défaut. Autrement dit, chaque instruction SQL est considérée comme une transaction individuelle qui est validée immédiatement. Notamment dans le cas des servlets, il peut être intéressant de gérer soi-même les transactions. Celles-ci se gèrent au moyen de l'objet qui implémente l'interface Connection. Pour cela, il faut tout d'abord désactiver le mode auto-commit par :

```
public abstract void setAutoCommit(boolean autoCommit) throws SQLException
```

Donc, ici, il nous faut programmer :

```
| setAutoCommit (false);
```

Le contrôle des transactions s'effectue alors tout naturellement au moyen des méthodes **commit()** et **rollback()** déjà évoquées.

A remarquer qu'en mode autocommit, si on n'effectue pas un select après une mise à jour update, il faut fermer la connexion pour que cette mise à jour soit prise en compte :

```
con.close();
```

♦ **les champs NULL** : on peut détecter qu'une méthode getXXX() a fourni une valeur NULL (inconnue) en utilisant la méthode de l'interface ResultSet :

```
public abstract boolean wasNull() throws SQLException
```

Donc, par exemple :

```
| double q = rs.getDouble("quantite");
| if (!rs.wasNull()) System.out.println("Quantité en stock = " + q);
```

♦ **les procédures stockées** : l'interface **CallableStatement**, analogue à PreparedStatement (dont il est d'ailleurs dérivé) permet de faire appel à des procédures PL/SQL stockées dans la base de données. Sans entrer dans les détails, disons simplement que l'on obtient un objet "procédure stockée" au moyen de la méthode de Connection :

```
public abstract CallableStatement prepareCall(String sql) throws SQLException
```

la chaîne de caractères attendue ayant l'une des deux formes standardisées :

```
"{call <nom_procédure>(? , ?)}"
"? = call <nom_procédure>(? , ?)"
```

selon que la procédure ne renvoie pas de valeur ou au contraire en fournit une. Donc, l'objet procédure viendra, par exemple, de :

```
| CallableStatement cstmt = con.prepareCall("{call recompense(?,?)}"
```

La suite est analogue aux instruction dynamiques, si ce n'est que les paramètres de sortie doivent être précisés, avec leur type, au moyen de :

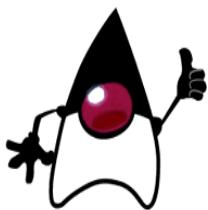
```
public abstract void registerOutParameter(int parameterIndex, int sqlType)  
throws SQLException
```

Par exemple :

```
| csmt.registerOutParameter(2, Types.FLOAT);
```

Le type est donc indiqué au moyen des variables de classe de la classe Types du package java.sql, par exemple :

```
public static final int FLOAT
```



N'importe qui peut donc à présent consulter une base de données et en connaître les détails les plus intimes ... N'importe qui ☺ ? Sans montrer patte blanche ? Est-ce bien normal ?

XIII. La cryptographie et Java



La vérité émerge plus facilement de l'erreur que de la confusion.

(F. Bacon)

1. Les caractéristiques idéales de la communication sécurisée

La **cryptologie** ("science du secret" en grec) est l'ensemble des techniques permettant d'assurer à des informations écrites une ou plusieurs des propriétés suivantes :

- ◆ la **confidentialité** des données : les données transmises doivent être incompréhensibles pour toute personne qui n'est pas l'émetteur ou le récepteur visé; c'est évidemment ici qu'interviennent les méthodes de **cryptographie** ("écriture secrète" en grec); par abus de langage, le terme de "cryptographie" est souvent utilisé en lieu et place de "cryptologie";
- ◆ l'**authentification** : celui qui reçoit le message doit être certain de l'origine de celui-ci; c'est ci qu'interviennent les notions de signature électronique;
- ◆ l'**intégrité** : celui qui reçoit le message doit être certain que les données contenues n'ont pas été modifiées depuis leur envoi par l'émetteur;
- ◆ la **non-répudiation** : l'émetteur ne doit pas pouvoir nier dans la suite avoir envoyé un tel message.

Bien sûr, qui dit espion dit contre-espionnage : la **crypto-analyse** est l'ensemble des techniques permettant de rendre clair le contenu d'un texte crypté. C'est une autre composante de la cryptologie.

Le **cryptage**, également appelé "**chiffrement**", est l'opération qui est la plus connue dans le grand public et c'est ce problème qui a historiquement été traité le premier. On le retrouve à l'heure actuelle dans les préoccupations informatiques, où l'on souhaite qu'un message passant sur le réseau ne puisse être compris au moyen d'un simple sniffer ...

2. Les messages secrets : un petit historique

Au-delà de l'intérêt purement culturel, un petit passage par un condensé de l'histoire de la cryptographie est bien utile pour en apprêhender les concepts principaux (pour des plus amples exposés sur cette histoire fascinante, on consultera les ouvrages cités en fin d'ouvrage : "The code book", "Dossier - Pour la Science" et "Science & Vie Junior").

Contentons-nous donc des points les plus marquants.

Des exemples de cryptographie sont recensés très tôt : vers 1900 av. J.C., un scribe égyptien codait des informations en utilisant des hiéroglyphes non standards, tandis qu'un scribe mésopotamien, vers 1500 av. J.C., codait en une formule chiffrée un procédé de fabrications d'enduits pour poteries.

2.1 Le codage de César et le principe de substitution

Traditionnellement cependant, on cite pour débuter le code de **César**, qui l'utilisa pendant la guerre des Gaules. Son principe est simple : chaque lettre d'un message est remplacée par celle qui se trouve x positions plus loin dans l'alphabet. Par exemple pour $x=3$:

caractère clair	a	b	c	d	e	f	...	x	y	z
caractère crypté	d	e	f	g	h	i	...	a	b	c

Si donc le message clair [*plain text*] est "vilvens", le texte crypté [*cipher text*] sera donné par :

message clair (plain text)	v	i	l	v	e	n	s
message crypté (cipher text)	y	l	o	y	h	q	v

Un tel cryptage est basé sur le principe de **substitution de caractères** : chaque lettre est remplacée par une autre lettre. On dira encore que **la clé du cryptage** est le nombre de positions, donc ici 3. Bien clairement, **une telle clé doit rester secrète**. D'autre part, cette clé est utilisée dans un **algorithme de cryptage** ("décaler les lettres de x positions" – le tableau ci-dessus en est l'illustration) et **cet algorithme de cryptage est public** (tout le monde peut le connaître): *c'est la connaissance de la clé qui est fondamentale*, pas celle de l'algorithme.

Un tel système présente des faiblesses évidentes :

- ♦ une lettre en remplace une autre : le message crypté est de la même longueur que le message original;
- ♦ une lettre donnée est toujours remplacée par la même lettre; une *analyse fréquentielle* peut alors permettre de deviner quelle lettre est cachée derrière une lettre donnée, pour peut que l'on sache dans quelle langue le message original a été écrit; en effet, si le message est écrit en français, on peut se baser sur l'analyse statistique des textes français pour obtenir le pourcentage relatif de chaque lettre :



- en utilisant le cryptage ci-dessus, on découvrira vite que le caractère "h" se reproduit le plus souvent et qu'il est donc probable qu'il représente en fait le caractère "e" !

Remarque

- 1) La clé qui a servi à crypter sert aussi pour décrypter.
- 2) On pourrait aussi imaginer que chaque lettre reste inchangée, mais que l'ordre des lettres soit modifié : c'est le principe de **transposition de caractères**. Ainsi, l'algorithme de cryptage pourrait être : "permuter les lettres de 3 couples successifs, laisser la suivante inchangée, puis recommencer" :

message clair (plain text)	c	l	a	u	d	e	v	i	l	v	e	n	s
message crypté (cipher text)	l	c	u	a	e	d	v	l	i	e	v	s	n

Les possibilités sont cependant plus réduites – mais on peut combiner transposition et substitution (voir plus loin).

2.2 Les clés secrètes plus élaborées

On a imaginé ensuite des clés plus difficiles à deviner qu'un simple nombre et donnant des substitutions plus compliquées à deviner (parce que sans logique arithmétique). Par exemple, la clé peut être une phrase comme "white is the sky" (phrase facile à retenir, pas à découvrir). L'algorithme de cryptage consiste alors à utiliser un tableau de substitution qui associe les premières lettres de l'alphabet aux lettres de la clé (sans répétition évidemment) :

clair	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
crypté	w	h	i	t	e	s	k	y	z	a	b	c	d	f	g	j	l	m	n	o	p	q	r	u	v	x

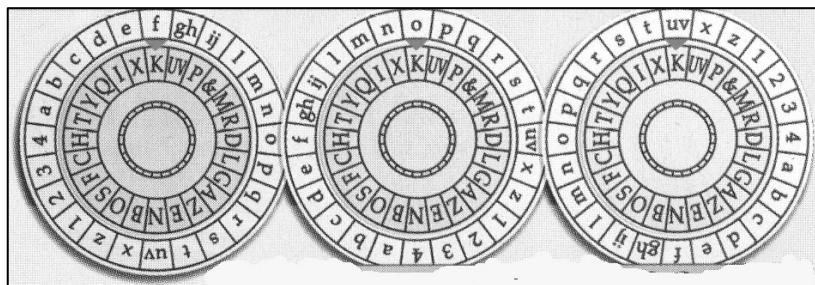
A nouveau, l'algorithme de cryptage est public ("on substitue en utilisant une phrase"), mais la clé doit rester secrète. Celle-ci doit être produite d'une certaine manière – par exemple, extraite d'un livre ou selon l'imagination du moment ou en fonction d'un sens caché : on peut donc parler ici d'un **algorithme de génération de clé**. Celui-ci peut être secret ou public.

On peut remarquer que l'analyse fréquentielle est toujours fort utile pour casser un tel système ...

2.3 Les cryptages de la Renaissance

Si le Moyen-Age n'a pas vraiment apporté de nouveautés, la Renaissance a par contre produit de nouvelles idées. Ainsi, on peut citer au crédit de cette époque :

- ♦ le principe des changements d'alphabets en cours de cryptage proposé par Leon Battista Alberti (1404-1472) : on parle encore de "**chiffrements polyalphabétiques**". L'idée : changer d'alphabet crypté en cours de cryptage d'un message. Pour la mise en œuvre, Alberti proposa le cadran chiffreur, qui met en correspondance l'alphabet clair (sur le disque extérieur du cadran) et l'alphabet crypté (sur le disque intérieur). Pour l'utiliser, il suffit de connaître une correspondance lettre claire/lettre cryptée (par exemple, f clair = k crypté → "f" est "calé" sur "k") pour disposer d'un système de cryptage :



Mais l'idée d'Alberti est de changer de "calage" à intervalles répétés. La clé est alors composée d'un mot dont les lettres donnent les calages successifs et d'un nombre indiquant le nombre de caractères que l'on peut coder avant de passer au calage suivant. Ainsi, par exemple, pour la clé "fou/5" :

message clair (plain text)	c	l	a	u	d	e	v	i	l	v	e	n	s
message crypté (cipher text)	q	&	t	n	i	c	d	y	q	d	e	c	i

↓ ↓ ↓

f calé sur k o calé sur k u calé sur k

Des tels disques ont été utilisés, par exemple, durant la guerre de Sécession américaine (1860-1866).

- ♦ la grille de Blaise de Vigenère (1523-1596) : celle-ci se présente sous la forme suivante

Plain	a b c d e f g h i j k l m n o p q r s t u v w x y z
1	B C D E F G H I J K L M N O P Q R S T U V W X Y Z A
2	C D E F G H I J K L M N O P Q R S T U V W X Y Z A B
3	D E F G H I J K L M N O P Q R S T U V W X Y Z A B C
4	E F G H I J K L M N O P Q R S T U V W X Y Z A B C D
5	F G H I J K L M N O P Q R S T U V W X Y Z A B C D E
6	G H I J K L M N O P Q R S T U V W X Y Z A B C D E F
7	H I J K L M N O P Q R S T U V W X Y Z A B C D E F G
8	I J K L M N O P Q R S T U V W X Y Z A B C D E F G H
9	J K L M N O P Q R S T U V W X Y Z A B C D E F G H I
10	K L M N O P Q R S T U V W X Y Z A B C D E F G H I J
11	L M N O P Q R S T U V W X Y Z A B C D E F G H I J K
12	M N O P Q R S T U V W X Y Z A B C D E F G H I J K L
13	N O P Q R S T U V W X Y Z A B C D E F G H I J K L M
14	O P Q R S T U V W X Y Z A B C D E F G H I J K L M N
15	P Q R S T U V W X Y Z A B C D E F G H I J K L M N O
16	Q R S T U V W X Y Z A B C D E F G H I J K L M N O P
17	R S T U V W X Y Z A B C D E F G H I J K L M N O P Q
18	S T U V W X Y Z A B C D E F G H I J K L M N O P Q R
19	T U V W X Y Z A B C D E F G H I J K L M N O P Q R S
20	U V W X Y Z A B C D E F G H I J K L M N O P Q R S T
21	V W X Y Z A B C D E F G H I J K L M N O P Q R S T U
22	W X Y Z A B C D E F G H I J K L M N O P Q R S T U V
23	X Y Z A B C D E F G H I J K L M N O P Q R S T U V W
24	Y Z A B C D E F G H I J K L M N O P Q R S T U V W X
25	Z A B C D E F G H I J K L M N O P Q R S T U V W X Y
26	A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Pour réaliser un cryptage, on utilise une simple clé, par exemple "white". On recopie alors cette clé le nombre de fois nécessaire en dessous du texte clair. Alors, la lettre de la clé fixe une ligne dans la grille, la lettre du message clair fixe la colonne : la lettre trouvée à l'intersection est la lettre cryptée :

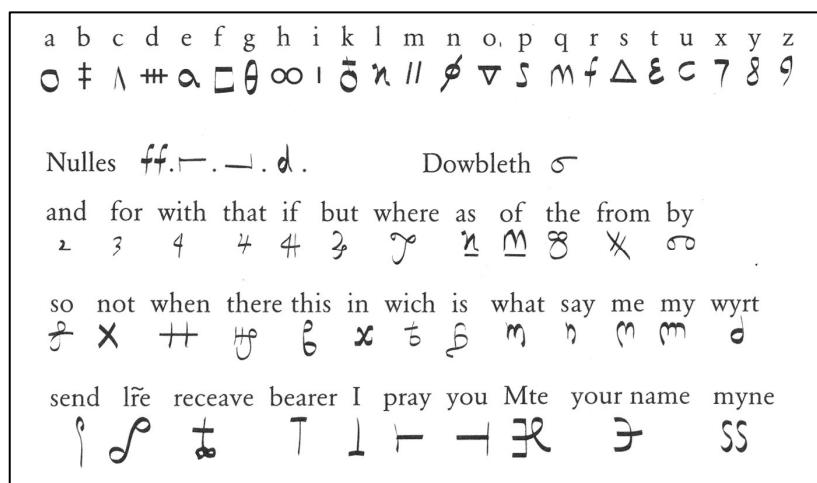
message clair (plain text)	c	l	a	u	d	e	v	i	l	v	e	n	s
clé	w	h	i	t	e	w	h	i	t	e	w	h	i
message crypté (cipher text)	y	s	i	m	h	a	c	q	e	z	a	u	a

Dans les deux cas, l'algorithme de cryptage est toujours public. Mais surtout, une lettre claire n'est pas forcément cryptée de la même manière à chaque occurrence (mais cela peut arriver ;-) !) : le cryptage d'un caractère dépend de sa position. Et l'analyse fréquentielle n'est plus daucun secours ☺ !

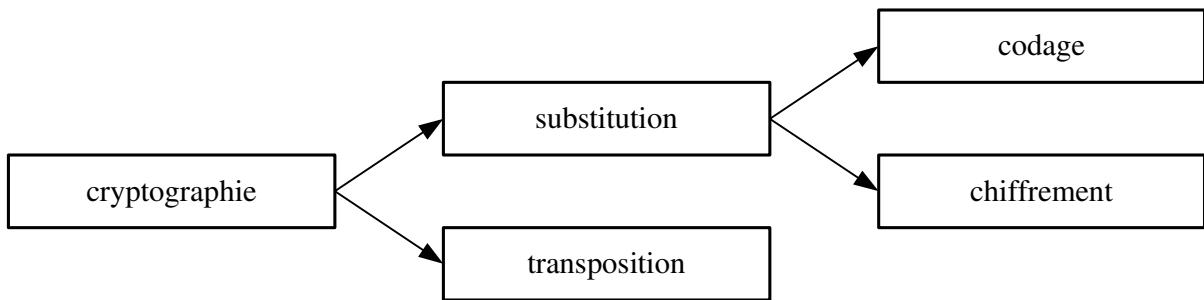
2.4 Cryptage et codage

La Renaissance et l'époque Classique ont par contre produit de nouvelles idées. Ainsi, on peut citer le système utilisé par Mary Stuart et ses alliés :

- ◆ les lettres y sont représentées par des caractères spéciaux;
- ◆ certains mots sont représentés par d'autres caractères spéciaux;
- ◆ certains caractères ne servent à rien d'autre qu'à tout embrouiller, car ils ne représentent rien !



Lorsque l'on en arrive à remplacer des mots par d'autres, on parle plus particulièrement de "**codage**" [code] alors que le remplacement de lettres par d'autres lettres (pas forcément toujours les mêmes) se désigne plutôt par le terme de "**chiffrement**" ou "**chiffre**" [cipher]. En résumé :



L'époque de Louis XIII et Louis XIV, par exemple, utilisa des codes où un mot clair était remplacé par une autre mot (par exemple, "Le Pape" devenait "La rose") ou par un nombre (par exemple, dans le "Grand Chiffre de Louis XIV, "faire" est codé 35, "faible" 67, etc). On ne peut pas ne pas citer un grand spécialiste des codes et décryptages : Antoine **Rossignol** (1600-1682), qui créa la "Grand Chiffre" de Louis XIV, lequel résistera pendant plus de 200 ans au décryptage !

2.5 L'utilisation combinée de la substitution et de la transposition

Le principe de transposition appliqué aux caractères n'est pas très riche en possibilités. Cependant, ce même principe peut devenir plus intéressant si on l'applique à des blocs de caractères. Ainsi, on peut imaginer de **découper le message clair en blocs de longueur fixe**, puis de transposer les blocs selon une certaine règle, laquelle utilise éventuellement une clé. En voici un exemple, celui de la transposition simple à clé : si "latin" est la clé, découpons le message "évacuer les bases" en blocs de 5 lettres et alignons ces blocs en colonnes :

<i>l</i>	<i>a</i>	<i>t</i>	<i>i</i>	<i>n</i>
----------	----------	----------	----------	----------

<i>e</i>	<i>v</i>	<i>a</i>	<i>c</i>	<i>u</i>
<i>e</i>	<i>r</i>	<i>l</i>	<i>e</i>	<i>s</i>
<i>b</i>	<i>a</i>	<i>s</i>	<i>e</i>	<i>s</i>

Réécrivons ensuite les lettres constituant la clé selon l'ordre alphabétique et permutions les colonnes en conséquence :

<i>a</i>	<i>i</i>	<i>l</i>	<i>n</i>	<i>t</i>
----------	----------	----------	----------	----------

<i>v</i>	<i>c</i>	<i>e</i>	<i>u</i>	<i>a</i>
<i>r</i>	<i>e</i>	<i>e</i>	<i>s</i>	<i>l</i>
<i>a</i>	<i>e</i>	<i>b</i>	<i>s</i>	<i>s</i>

Le message crypté pourrait être la concaténation des 3 blocs obtenus. Mais on peut encore raffiner en parcourant la grille en boustrophédon (comme le bœuf qui tire la charrue pour les labours) : on descend la 1^{ère} colonne pour remonter la 2^{ème} et ainsi de suite. Le message crypté est donc finalement "vraeeceebssuals".

On peut remarquer cependant que l'utilisation de blocs apporte un nouveau problème. En effet, avec le même algorithme et la même clé, supposons vouloir crypter "vilvens sera absent". Cela donne pour démarrer

<i>l</i>	<i>a</i>	<i>t</i>	<i>i</i>	<i>n</i>
----------	----------	----------	----------	----------

v	i	l	v	e
n	s	s	e	r
a	a	b	s	e
n	t			

On voit immédiatement que le dernier bloc est incomplet. On pourrait penser le compléter avec un caractère comme "0" ou "&" – mais ce serait donner une information au cryptoanalyste qui tenterait de décoder ! Il faudra donc un **algorithme de remplissage [padding]** qui génère des caractères de remplissage qui ne soient pas détectables trop facilement.

La première guerre mondiale a vu ce genre de cryptage utilisé en version améliorée par les Allemands : ils combinèrent en fait les principes de substitution de caractères et de transposition de blocs pour créer le code ADFGX. Il doit son nom au fait que le point de départ de sa mise en œuvre est un *carré de Polybe* construit sur ces 5 lettres. Un carré de Polybe est un tableau 5x5 du type suivant :

	1	2	3	4	5
1	a	b	c	d	e
2	f	g	h	i,j	k
3	l	m	n	o	p
4	q	r	s	t	u
5	v	w	x	y	z

qui implique un cryptage pour lequel, par exemple, "d" est codé en "14". On peut constater que nombre de symboles utilisés pour le codage est réduit, ce qui rend son analyse plus difficile. Bien sûr, on peut imaginer que le carré est rempli au moyen d'une clé pour le début, puis par ordre alphabétique pour les lettres restantes. Evidemment, deux caractères partagent le même code (ici, "i" et "j" – ce pourrait être "u" et "v").

Dans le cas du cryptage allemand de 14-18, les chiffres sont remplacés par les 5 lettres ADFGX – d'où son nom :

	A	D	F	G	X
A	c	q	h	b	l
D	o	e	i,j	s	g
F	d	n	k	v	y
G	t	f	x	r	z
X	m	w	u	p	a

Si nous voulons crypter "vilvens sera absent" avec comme clé "latin" :

msg clair (plain text)	v	i	l	v	e	n	s	s	e	r	a	a	b	s	e	n	t
msg crypt é (ciph)	F G	D F	A X	F G	D D	F D	D G	D G	D D	G G	X X	X X	A G	D G	D D	F D	G A

er text)													
-------------	--	--	--	--	--	--	--	--	--	--	--	--	--

Mais le cryptage ne s'arrête pas là. Le message crypté (sous forme de *bigrammes* = lettres doubles) est ensuite rangé dans une grille :

<i>l</i>	<i>a</i>	<i>t</i>	<i>i</i>	<i>n</i>
----------	----------	----------	----------	----------

FG	DF	AX	FG	DD
FD	DG	DG	DD	GG
XX	XX	AG	DG	DD
FD	GA			

puis nous réécrivons ensuite les lettres constituant la clé selon l'ordre alphabétique et permutions les colonnes en conséquence (il s'agit donc d'une transposition sur les blocs) :

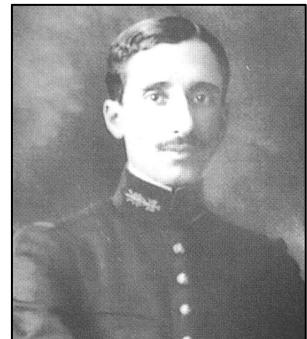
<i>a</i>	<i>i</i>	<i>l</i>	<i>n</i>	<i>t</i>
----------	----------	----------	----------	----------

DF	FG	FG	DD	AX
DG	DD	FD	GG	DG
XX	DG	XX	DD	AG
GA		FD		

Le message crypté sera donc finalement :

DF FG FG DD AX DG DD FD GG DG XX DG XX DD AG GA FD

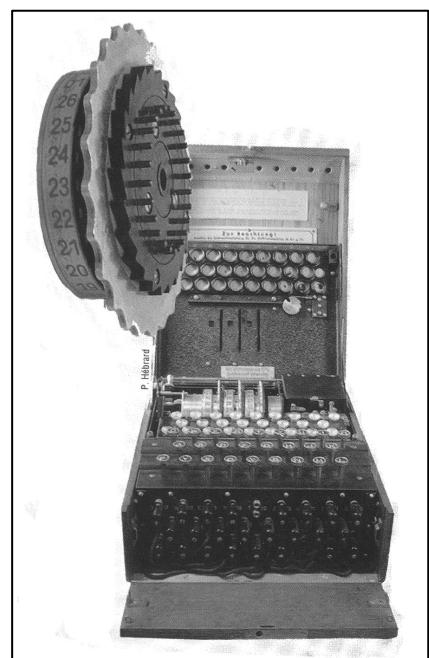
C'est le grand mérite du capitaine Georges **Painvin** (1886-1980) d'être parvenu à casser un tel code !



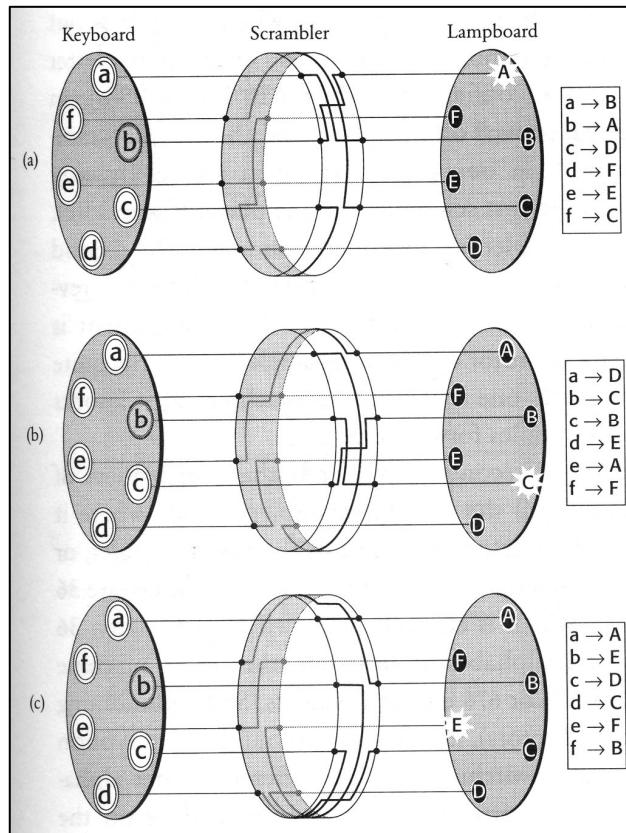
2.6 Le cryptage par procédé électromécanique : Enigma

Après la 1^{ère} guerre mondiale, les armées européennes commencent à se doter de machines destinées à automatiser les opérations de chiffrement-déchiffrement. L'armée allemande, encore elle, va s'équiper d'une amélioration d'une machine postale nommée **Enigma**. Elle reprend l'idée des disques de chiffrement d'Alberti, mais avec de considérables améliorations :

1) L'élément de base est un disque métallique comportant 26 contacts électriques en entrée et 26 contacts en sortie. Chaque entrée est câblée à une sortie – bien sûr, le schéma de ce câblage est secret, et il existe d'ailleurs en différentes versions. Le destinataire et l'émetteur se sont mis d'accord sur le calage de départ. Si l'expéditeur appuie sur "b", par exemple, le câblage électrique conduit, par exemple, à "a" (ce que l'opérateur saura par la lampe allumée sur la machine). L'appui sur la touche a cependant eu un autre effet : le disque a tourné d'une position, modifiant tout le câblage ! Un nouvel appui sur



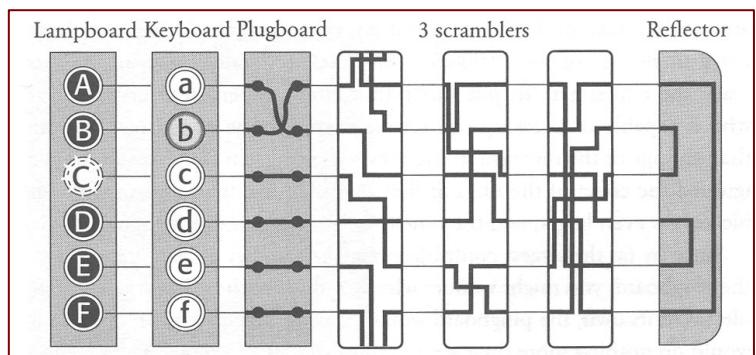
la touche "b" conduit à présent à "c", avec en corollaire un nouveau déplacement du disque d'une position. Etc.



2) En réalité, le disque de codage n'est pas seul : il est suivi de deux autres disques au câblage interne différent (il y avait donc initialement 3 types de disques - dans la suite, 5 types devinrent possibles !). Le courant électrique ne repasse donc pas directement vers le tableau de lampe, mais est transmis du 1^{er} au 2^{ème} disque, puis du 2^{ème} au 3^{ème} disque. En fait, au bout de 26 décalages du 1^{er} disque, un décalage du 2^{ème} disque va s'opérer. Et, beaucoup plus tard, au bout de 26 décalages du 2^{ème} disque, un décalage du 3^{ème} disque va s'opérer !

3) De plus, un réflecteur renvoie le courant dans l'autre sens dans le jeu des 3 disques, pour enfin allumer une lampe parmi les 26 possibles.

4) L'initialisation de la machine, outre la position de départ de chacun des trois disques, faisait encore intervenir un paramètre supplémentaire : six câbles permettaient d'intervertir 6 couples de lettres (l'opérateur appuyait par exemple sur "a", mais le signal électrique généré correspondait par exemple à un "g"), les autres lettres restant dans leur câblage initial.



Quand on y réfléchit, le nombre de configurations possibles d'une machine Enigma à trois rotors était donc de :

- $26 \times 26 \times 26$ positions initiales des 3 disques;

- toutes les permutations possibles des 3 disques (donc $3! = 6$);
 - toutes les combinaisons possible de connexions de 6 paires de deux lettres.
- Le nombre de possibilités est l'ordre du milliard de milliards ;-)!

Bien sûr, le challenge des Alliés durant la 2^{ème} guerre mondiale fut de casser le système Enigma. A cette fin, on rassembla en Angleterre, dans le château de **Bletchley Park** (siège du service britannique du Chiffre), tous ceux qui pouvaient contribuer à cet objectif : mathématiciens, linguistes dont des égyptologues, joueurs de bridge, cruciverbistes recrutés par voie de petites annonces dans les journaux, joueurs d'échecs, ... Parmi eux, un génie des mathématiques : Alan Turing (1912-1954).



En fait, les Britanniques et leurs alliés disposaient des travaux des membres du bureau du Chiffre polonais. Ces derniers étaient parvenus à décrypter les messages codés en Enigma jusqu'à l'invasion de leur pays par les Allemands (en 1939), ceci grâce à leurs talents mathématiques d'une part et aussi grâce à la mise au point d'une machine nommée "Bomba" : celle-ci comportait 6 répliques d'Enigma, chacune configurée avec l'un des arrangements possibles des trois rotors. Face à un message crypté, la machine envisageait toutes les possibilités : en 1938, la clé pour un message donné était trouvée dans un temps moyen de deux heures. Quand les Allemands améliorèrent leur système en choisissant 3 disques parmi 5 possibles, le nombre de configuration d'Enigma possibles passa à 60 : une machine "bombe" adaptée devenait impossible à construire. Turing parvint dans la suite cependant à décrypter les messages ennemis, en utilisant la technique de "recherche du mot probable" (ainsi, un message météo avait beaucoup de chance de commencer de manière standard et de contenir le mot "Wetter"). De plus, un autre mathématicien, Gordon Welchman, mis au point une bombe améliorée.

Les Alliés furent ainsi capables de décrypter les messages de la Wehrmacht et la Luftwaffe – mais pas ceux de la Kriegmarine (et de ses féroces sous-marins), qui utilisaient une nouvelle Enigma M4 munie d'un réflecteur se comportant, en partie, comme un 4^{ème} rotor, démultipliant ainsi le nombre de possibilités. La capture de l'U-Boot 559 et de sa documentation d'emploi de l'Enigma M4 fut évidemment un événement décisif : les Alliés purent ainsi construire des bombes adaptées et aussi des machines Colossus qui permirent le décryptage du code du haut commandement allemand.

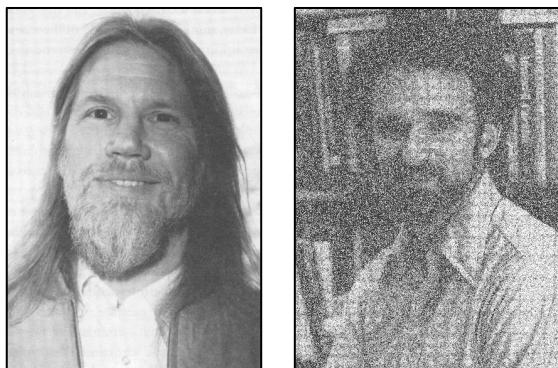
On connaît l'issue de la 2^{ème} guerre mondiale ...

2.7 Le protocole de Diffie-Hellman : la clé partagée

Dans tout ce qui a été dit jusqu'à présent, on a supposé que les deux parties de la communication cryptée étaient en possession de la clé, secret fondamental garant de la confidentialité. En pratique, cependant, cette hypothèse n'est pas forcément facile à satisfaire

– d'autant que l'on peut obtenir la clé du destinataire potentiel si on la lui demande avec conviction (= torture par exemple). En particulier, lorsque les deux parties sont aux deux extrémités d'un réseau informatique, on peut craindre, avec raison, que la simple transmission de la clé par ce réseau soit l'objet d'une prise par un hacker armé d'un sniffer !

Tout le problème revient donc au partage exclusif de la clé. Et c'est ici qu'interviennent deux génies des années 1970 : Whitefield **Diffie** (1944-) et Martin **Hellman** (1946-), rejoints en cours de réflexion par Ralph **Merkle** (1952-). Ils ont élaboré en 1976 un protocole d'échange entre deux parties leur permettant de s'accorder sur une clé secrète (donc connue d'eux seuls) sur base d'une discussion publique (donc qui peut être écoutée par n'importe qui, y compris les hackers !). Ce protocole, basée sur l'arithmétique modulaire et qui sera décrit plus loin, faisait ainsi intervenir les concepts de clé privée et clé publique.

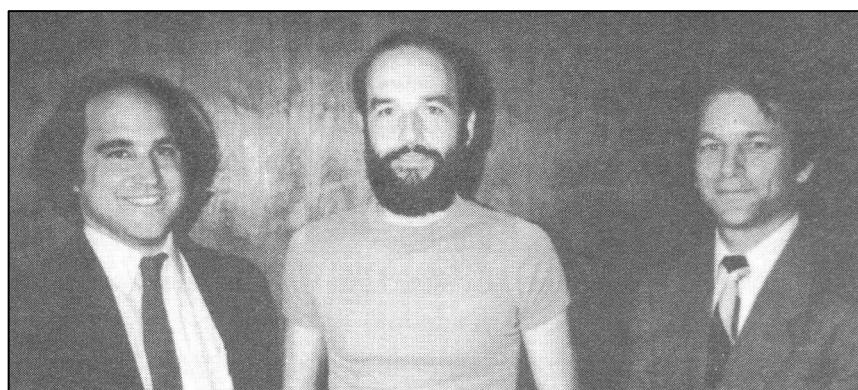


2.8 La cryptographie à clé publique

Ces concepts de clé privée - clé publique trouvèrent une implémentation concrète dès 1978 avec le système RSA, du nom de ses inventeurs (**Ron Rivest**, **Adi Shamir** & **Leonard Adleman**). Ce système de communication sécurisée, basé sur la difficulté de factoriser les grands nombres, consiste à

- ◆ crypter un message par un expéditeur avec la clé publique (donc connue de tous) du destinataire;
- ◆ déchiffrer le message par son destinataire qui utilise la clé privée associée à la clé publique qui a servi à crypter et qu'il est le seul à posséder.

Autrement dit, **la clé qui sert à crypter n'est pas celle qui sert à déchiffrer** ! Nous y reviendrons plus loin.



Ainsi éclairés par l'Histoire (qui n'est pas finie : courbes elliptiques, cryptographie quantique, etc), nous pouvons à présent devenir plus techniques ...

3. Les bases des techniques cryptographiques

3.1 Clés et algorithmes

Classiquement, le schéma d'une transmission codée est le suivant :

- ◆ rédaction du **texte en clair** (en anglais, *plaintext* ou *cleartext*);
- ◆ production d'un texte codé ou **texte chiffré** (en anglais, *ciphertext*); cette opération s'appelle le chiffrement (en anglais, *encryption*) et la méthode utilisée s'appelle un chiffre (en anglais, *cipher*); **l'algorithme de chiffrement** de la méthode utilise le plus souvent une ou deux **clés**, sans la(les)quelle(s) il est impossible de reconstruire le message original;
- ◆ transmission par le réseau;
- ◆ déchiffrement par le récepteur, toujours un utilisant l'une ou l'autre clé.

La notion de clé est donc fondamentale. D'après l'historique, nous pouvons dire que

une clé est un bloc d'informations permettant un chiffrement ou un déchiffrement.

et cette clé peut être produite par **un algorithme de génération de clé**. En ce qui concerne la méthode de production du texte chiffré, on peut distinguer plus précisément :

- ◆ **les algorithmes symétriques ou à clé secrète** : *la même clé est utilisée au chiffrement et au déchiffrement*; un exemple plus qu'élémentaire est de remplacer chaque lettre du message par celle qui se trouve x positions plus loin dans l'alphabet – dans ce cas, la clé est la valeur x de ce nombre de positions;
- ◆ **les algorithmes asymétriques ou à clé publique** : ici, *on utilise une paire de clés*, soit *la clé publique* (en anglais, *public key*) et *la clé privée* (en anglais, *private key*), l'une servant au chiffrement et l'autre au déchiffrement sans que le même clé puisse jouer les deux rôles; ils sont utilisés pour assurer la confidentialité (donc, le chiffrement) et l'authentification (c'est-à-dire les signatures électroniques) – on parle encore de **PKI** (**P**ublic **K**ey **I**nfrastructure).

3.2 Les modes de chiffrement

De plus, on peut encore, selon l'un ou l'autre algorithme, distinguer la manière dont les blocs d'un texte clair correspondent à des blocs de texte chiffré : on parle alors de "**mode de chiffrement**".

1) Le chiffrement de bloc [*block cipher*] *transforme un bloc de données claires d'une taille préalablement fixée en un bloc de données chiffrées de même taille*. Pour chiffrer un message, il faut donc

- a) découper celui-ci en blocs de taille acceptée par l'algorithme utilisé;
- b) chiffrer chaque bloc : ceci se fait en utilisant un mode chaînage qui peut être l'un des modes de chiffrement courants :

- ◆ **ECB** (**E**lectronic **C**ode **B**ook) : *un bloc de texte clair se chiffre en un bloc de texte chiffré, indépendamment des autres blocs*; il faut remarquer que deux blocs identiques produisent les mêmes blocs chiffrés et qu'il y a peu de protection sur l'intégrité du message, puisque l'indépendance des blocs chiffrés ne permet pas de détecter des permutations, duplications ou suppressions de blocs;

- ♦ **CBC (Cipher Bloc Chaining)** : chaque bloc de texte clair est combiné par un XOR avec le bloc de texte chiffré précédent (un "vecteur d'initialisation" – IV pour les intimes - fournit le bloc chiffré pour le premier bloc clair); cette fois, deux blocs identiques ont peu de chance de produire les mêmes blocs chiffrés et il y a protection sur l'intégrité du message, puisque des permutations, duplications ou suppressions de blocs auront des implications sur les blocs résultants.

Evidemment, le message à coder sera le plus souvent d'une taille non multiple de la taille du bloc type de traitement; il faudra donc compléter le message avec des caractères de remplissage (*padding*). Un principe des plus simples est de remplir les bytes non occupés par le nombre de ces bytes non occupés. Les paddings les plus connus sont ceux des recommandations **PKCS#5** (algorithme symétrique DES) ou **PKCS#7** (algorithme asymétrique RSA) – les **PKCS** (Public Key Cryptography Standards) sont un ensemble de spécifications standards publiées par RSA Data Security (ces standards sont numérotés de 1 à 15 – voir l'annexe en fin d'ouvrage).

On peut encore épingle que, dans un chiffrement de bloc, le nombre de blocs chiffrés est le même que celui de blocs clairs – c'est un peu la faiblesse du système.

2) Dans le contexte des chiffrements symétriques, le **chiffrement de flux** [*stream cipher*] opère sur les bits, ou du moins sur des unités plus petites comme les bytes, et les transforme de manière variable selon le moment du chiffrement.

Le principe est de combiner le texte clair avec une clé aléatoire [*key stream*], de même longueur et qui ne sert qu'à un seul cryptage (on parle encore de "**one-time pad**" ou OTP – le chiffrement de Vernam est apparenté); la transformation opérée sur les caractères du texte clair varie donc au cours du temps. Un tel cryptage a été démontré mathématiquement incassable.

Cependant, en pratique, il est fort malaisé de générer à chaque transmission une clé aléatoire de longueur quelconque; les algorithmes réellement utilisés se basent sur une clé de taille plus réduite (par exemple, 128 bits), qui permet de *générer une clé pseudo-aléatoire qui sera effectivement utilisée pour le cryptage*. Malheureusement, l'inviolabilité du cryptage n'est plus assurée et une clé trop courte donne un cryptage qui peut éventuellement être cassé "par force brute" (c'est-à-dire par essais successifs).

4. Les chiffrements symétriques

4.1 Le principe général

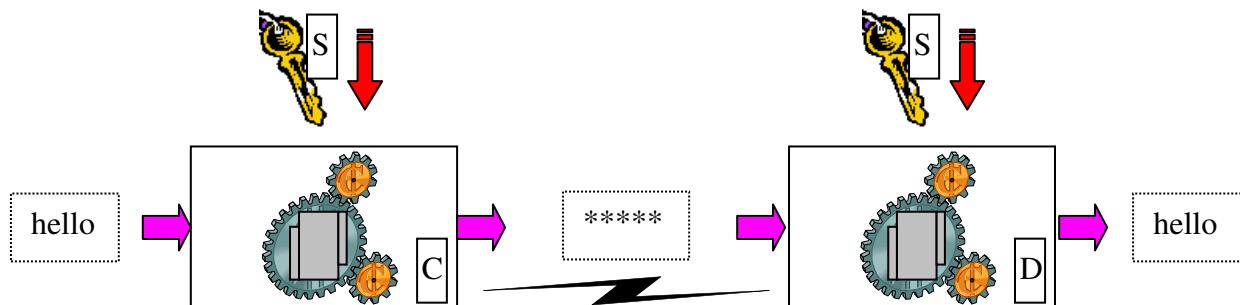
Un algorithme symétrique est celui auquel on pense immédiatement dans un contexte de cryptage : il utilise la même clé pour chiffrer et déchiffrer un message. Bien sûr, le principal danger est que la clé soit interceptée, car tout qui la possède est capable de déchiffrer le message. Comme cette clé doit donc rester secrète, on parle encore d'algorithme à clé secrète.

4.2 Un exemple

Supposons que le message à transmettre soit simplement "hello". L'expéditeur (James) et le destinataire (Francesca) se mettent d'accord sur une clé secrète (disons "sydney"). Cette clé est utilisée par l'algorithme de chiffrement qui est ici remarquablement simple : il applique un XOR sur le code ASCII de chaque lettre du message (104-101-108-108-111) avec celui d'une lettre de la clé.

chiffrement :

message clair	"hello"	0110 1000	0110 0101	0110 1100	0110 1100	0110 1111
clé	"sidney"	0111 0011	0110 1001	0110 0100	0110 1110	0110 0101
XOR						
message crypté	*****	0001 1011	0000 1100	0000 1000	0000 0010	0000 1010



(S=clé Secrète; C=Chiffrement; D=Déchiffrement)

déchiffrement :

message crypté	*****	0001 1011	0000 1100	0000 1000	0000 0010	0000 1010
clé	"sidney"	0111 0011	0110 1001	0110 0100	0110 1110	0110 0101
XOR						
message obtenu	"hello"	0110 1000	0110 0101	0110 1100	0110 1100	0110 1111

[Note : *Un grand merci à L.Herbiet pour les exemples de ce type qu'elle a patiemment concoctés ☺*]

4.3 Quelques algorithmes symétriques courants

On peut citer :

- ◆ **DES (Data Encryption Standard)** : c'est le standard du gouvernement américain depuis 1977. Il utilise par défaut des blocs de 64 bits et une clé secrète de 56 bits (plus précisément, la clé est de 64 bits, mais 8 d'entre eux servent à un contrôle de parité vérifiant l'intégrité de la clé; de plus, les bits de la clé représentent en fait 16 sous-clés de 4 bits). Il a été certifié par le NIST (National Institute of Standards and Technology) jusqu'en 1993.
- ◆ **TripleDES** : comme son nom l'indique, on y utilise trois fois de suite le DES en utilisant de plus une deuxième clé secrète. Les clés ont une taille de 128 ou 192 bits.
- ◆ **Blowfish** (B.Schneier) : spécialement développé pour les machines 32 bits, il utilise des blocs de 64 bits et une clé secrète de taille variable (<=448 bits); en fait, cette clé est utilisée pour générer des sous-clés qui permettront chacune un tour de chiffrement. Blowfish est significativement plus rapide que DES.
- ◆ **IDEA (International Data Encryption Algorithm)** : spécialement conçu pour résister aux techniques de crypto-analyse évoluées, il utilise des blocs de 64 bits, une clé secrète de 128 bits et 8 tours. Sa vitesse est du même ordre que celle de DES.
- ◆ **SAFER (Secure And Fast Encryption Routine)** : cet algorithme de chiffrement de bloc, très résistant semble-t-il, utilise des blocs de 32 bits, une clé de 64 ou 128 bits et un nombre de tour variable (jusqu'à 10).

- ◆ **Rijndael** (d'après le nom des concepteurs belges, V. Rijmen et J. Daemen) et **AES** (Advanced Encryption Standard) : c'est l'algorithme sélectionné par le NIST comme successeur de DES; Rijndael utilise des clés multiples de 32 bits dans l'intervalle [128,256]; AES en est un sous-ensemble se limitant à des clés de 128, 192 ou 256 bits.
- ◆ **RC** (Rivest Cipher) : développé, et gardé secret, par la société américaine RSA Data Security, il utilise des blocs avec une clé de taille variable et est réputé plus rapide que DES. En fait, il existe plusieurs déclinaisons de cet algorithme : RC2 (pour des clés de 40 bits maximum) et RC5, entièrement paramétrable quant à la taille du bloc (32, 64 et 128 bits), la taille de la clé (jusqu'à 2048 bits) et le nombre de tours (jusqu'à 255). **RC4** est un algorithme de chiffrement de flux qui utilise des clés de 40, 128 ou 256 bits. Il est à la base des chiffrements utilisés dans **SSL** (Secure Sockets Layer – celui de https) et dans le cryptage **WEP** (Wired Equivalent Privacy) qui a pour but de sécuriser les communications WIFI (Wireless Fidelity) c'est-à-dire les réseaux sans fil (mais il tend à être remplacé par WPA - Wireless Protected Access).

4.4 Les générateurs de clés

Comme de nombreux algorithmes de chiffrement symétrique (et asymétrique d'ailleurs aussi) sont **publics**, leur fonctionnement n'est pas un mystère et leur efficacité repose sur la complexité des clés utilisées. Un algorithme de chiffrement sera donc accompagné d'un algorithme de génération de clés destinées à son usage. Dans le cas de DES, par exemple, la génération de la clé à 64 bits repose sur un cheminement complexe appliquant des permutations, des décalages et des recombinaisons successifs.

Remarque

On appelle "clés faibles" (*weakkeys*) des clés qui conduisent à des chiffrements faciles à déchiffrer. On possède la liste de ces "mauvaises clés" pour les algorithmes comme DES ou TripleDES.

5. Le protocole de Diffie-Hellman

5.1 La base mathématique

L'arithmétique modulaire repose sur une idée simple. En effet, on associe à deux nombres entiers positifs x et y le reste de la division de x par y (encore appelé " x modulo y ") et on note :

$$x \% y = z \quad \text{ou} \quad x = z \bmod y$$

Par exemple,

$$15 \% 4 = 3 \text{ et } 15 \% 5 = 0$$

Dans la vie courante, nous travaillons en modulo 12 sans même y prêter attention. Ainsi, s'il faut laisser la pâte reposer 8h et qu'il est 9h du matin, nous en concluons que nous pourrons poursuivre la recette à $(9+8)\%12=5$ h du soir !

Mais l'intérêt du modulo dans le contexte de la cryptographie repose évidemment ailleurs :

a) d'une part, il permet d'associer à un nombre de valeur quelconque un nombre limité à une fourchette; par exemple, modulo 7 ne donnera jamais qu'un résultat compris entre 0 et 6.

b) d'autre part, l'opération modulo permet de simuler des comportements erratiques, aléatoires; ainsi, si nous considérons la fonction puissance de base 3 en arithmétique classique et en arithmétique modulaire 7 (par exemple), on constate que :

x	1	2	3	4	5	6
3^x	3	9	27	81	243	729
$3^x \% 7$	3	2	6	4	5	1

Autrement dit, une fonction croissante en arithmétique classique devient erratique en arithmétique modulaire.

c) La fonction $f(x): x \rightarrow x \% n$ est une **fonction non réversible**, c'est-à-dire que l'on ne peut retrouver un seul x à partir de la valeur de la fonction (ainsi, $15 \rightarrow 15 \% 4 = 3$, mais $27 \rightarrow 27 \% 4 = 3$ aussi $\rightarrow 3$ est l'"image" d'une infinité de nombres).

5.2 L'algorithme

Voyons à présent comment James et Francesca vont procéder pour arriver à partager une clé sans la faire passer sur le réseau.

1) Ils se mettent d'accord publiquement sur deux nombres n et p , n étant inférieur à p . Par exemple, ils conviennent que $p=11$ et $n=7$. Il s'agit en fait des deux paramètres d'une fonction puissance en arithmétique modulaire : $n^x \% p \rightarrow 7^x \% 11$.

2) James choisit un nombre aléatoire $A=3$ (par exemple), nombre qu'il gardera secret. Francesca choisit de même un nombre $B=6$, qu'elle gardera tout aussi secret.

3) James calcule $\alpha = 7^A \% 11 = 7^3 \% 11 = 343 \% 11 = 2$ - c'est sa clé publique et James l'envoie à Francesca.

De même, Francesca calcule $\beta = 7^B \% 11 = 7^6 \% 11 = 117649 \% 11 = 4$ - c'est sa clé publique et Francesca l'envoie à James.

Les deux correspondants disposent donc à présent chacun des deux clés publiques.

4) James utilise la clé publique β de Francesca pour construire une **clé secrète** selon

$$K_{\text{James}} = \beta^A \% 11 = 4^3 \% 11 = 64 \% 11 = 9.$$

De manière similaire, Francesca utilise la clé publique α de James pour construire une **clé secrète** selon

$$K_{\text{Francesca}} = \alpha^B \% 11 = 2^6 \% 11 = 64 \% 11 = 9.$$

Miracle : ils ont fabriqué la même clé, soit 9 ☺ ! Ils ne leur reste plus qu'à l'utiliser dans un cryptage symétrique.

Mais comment peut-on être sûr qu'un hacker ne peut pas, lui aussi, construire la même clé avec les éléments publics échangés ? Parce qu'il faut connaître ou A ou B , qui sont restés secrets et qu'il très difficile (c'est-à-dire impossible dans un temps raisonnable) de deviner $A(B)$ à partie de $\alpha(\beta)$ car la fonction $\%$ est une fonction non réversible.

6. Les chiffrements asymétriques

6.1 Le principe général

Le chiffrement symétrique, à une seule clé secrète, est sans doute le plus intuitif, mais souffre d'une faiblesse de taille : la clé doit absolument rester secrète, c'est-à-dire connue des seuls émetteurs et récepteurs. Si la clé tombe entre des mains étrangères, celles-ci n'auront aucune difficulté à déchiffrer le message. Or, pour des utilisateurs se trouvant parfois séparés par de grandes distances, il est clair qu'il faudrait faire circuler la clé secrète sur le réseau, ce qui est fort naïf et délicat ...

C'est pour cette raison que les scientifiques se sont mis à la recherche d'une autre méthode. Nous avons vu que Diffie et Hellman ont jeté les bases de la cryptographie à clé publique. Et c'est ce même Diffie qui poussa plus loin le raisonnement pour définir les bases du chiffrement asymétrique. L'idée maîtresse en est que chaque utilisateur dispose en fait de deux clés :

- ◆ **une clé publique**, qui peut donc être connue de tous; la seule contrainte est qu'elle soit accompagnée d'un certificat provenant d'une autorité en laquelle l'émetteur et le récepteur ont confiance;
- ◆ **une clé privée**, que l'utilisateur peut générer sans devoir la faire circuler sur le réseau – c'est évidemment ici qu'apparaît le caractère sécurisé.

Ainsi, si James souhaite envoyer un message à Francesca, il recherche la clé publique de celle-ci. James utilise cette clé pour chiffrer son message, puis le transmet à Francesca sur le réseau. Celle-ci n'a plus qu'à décoder le message avec sa clé privée. Autrement dit, ***n'importe qui peut envoyer un message à Francesca, mais elle seule peut le déchiffrer.*** Bien sûr, si Francesca désire répondre à James, elle procédera de même avec la clé publique de James, qui déchiffra avec sa clé privée.

A priori, cela peut sembler impossible à réaliser en pratique : et pourtant, tout se base sur le fait que, si il est facile de multiplier deux nombres premiers l'un par l'autre, il est, par contre, extrêmement difficile de retrouver deux tels nombres à partir d'un nombre donné (on parle encore de "factorisation des nombres premiers"). De manière plus générale, les algorithmes asymétriques se basent, outre sur la factorisation, sur l'arithmétique modulaire et les logarithmes discrets. Les exemples ci-dessous vous nous révéler ce mystère ...

6.2 Un exemple

Supposons à nouveau que le message à transmettre soit simplement "hello". James, l'expéditeur se met à la recherche de la clé publique de Francesca (la destinatrice). Cette clé publique (comme d'ailleurs la clé secrète de Francesca) se présente sous la forme d'un couple de nombres (n, e) qui permettent l'application de l'algorithme de chiffrement (de type RSA – voir ci-dessous) :

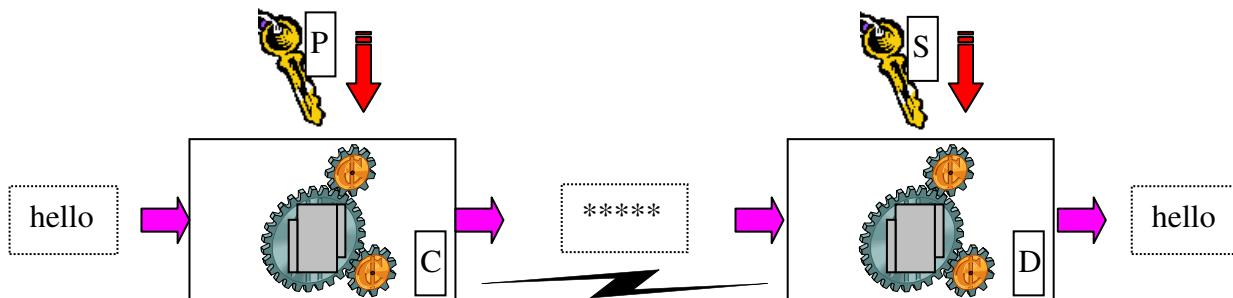
$$\text{<caractère chiffré>} = (\text{<caractère à chiffrer>}^e) \% n$$

Si donc la clé publique de Francesca est (3233, 17), alors James peut chiffrer son message **en utilisant cette clé publique** :

chiffrement :

message clair	"hello"	104	101	108	108	111
clé publique de Francesca (destinataire) = (3233, 17)						
c ¹⁷ % 3233						

message crypté	*****	2170	1313	745	745	2185
----------------	-------	------	------	-----	-----	------



(P=clé Publique; S=clé privée, qui doit rester Secrète)

Francesca déchiffrera le message reçu en utilisant le même algorithme **mais au moyen de sa clé privée** (qui doit donc rester secrète pour tout le monde), qui se présente également sous forme d'un couple de nombres (n,d), disons ici (3233,2753) :

déchiffrement :

message crypté	*****	2170	1313	745	745	2185
clé privée de Francesca (destinataire) = (3233, 2753)						
c ²⁷⁵³ % 3233						

message obtenu	"hello"	104	101	108	108	111
----------------	---------	-----	-----	-----	-----	-----

Bien sûr, rien ne prouve que c'est bien James qui a envoyé le message : c'est le problème de l'authentification et des signatures digitales ...

6.3 Quelques algorithmes asymétriques courants

On peut citer :

- ◆ **RSA** (R.Rivest, A.Shamir & L.Adleman) : basé sur la difficulté de factoriser les grands nombres, c'est *l'algorithme le plus utilisé au monde*; il est considéré comme efficace avec des clés de 1024 bits. Nous avons vu ci-dessus la formule de cryptage

$$\text{<caractère chiffré>} = (\text{<caractère à chiffrer>}^e) \% \text{n}$$

et de décryptage

$$\text{<caractère chiffré>} = (\text{<caractère à chiffrer>}^d) \% \text{n}$$

Comment les composants des couples (n,e) et (n,d) formant respectivement les clés publiques et privées sont-elles calculées ? Au moyen de la théorie des nombres premiers ! Plus précisément :

* **clé publique** : On choisit (aléatoirement) deux nombres premiers **p** et **q** relativement grands (pour notre exemple, $p=61$ et $q=53$). Alors, **n** (le *module*) est simplement donné par leur produit :

$$n = p \cdot q = 61 \cdot 53 = 3233$$

C'est ici que se situe la difficulté de celui qui voudrait percer le code : il faut tenter de factoriser le module ... L'*exposant public* (**e**) est choisi comme étant un entier dans $[3, n-1]$ (donc inférieur à n) et premier avec $z = (p-1) \cdot (q-1) = 60 \cdot 52 = 3120$; comme $3120 = 2^5 \cdot 3 \cdot 5 \cdot 13$, tout nombre non multiple de 2, 3, 5 et 13 peut convenir; $e=17$ est donc un candidat valable (l'algorithme ne réclame pas forcément un nombre premier – 49 eût donc pu convenir); en pratique, **e** est souvent pris égal à 3 ou encore 65537 si la valeur de **n** le permet.

* **clé privée** : Le module est le même. L'*exposant privé* **d** est un nombre tel que $(e \cdot d - 1)$ soit divisible par **z** (autrement dit, $e \cdot d \equiv 1 \pmod{z}$); donc $d=2753$ peut convenir, puisque $17 \cdot 2753 - 1 = 46800$ et est divisible par 3120 (résultat = 15).

En pratique, *les tailles des clés valides ne sont pas quelconques* : elles doivent être comprises dans l'intervalle $[384, 16384]$ et, de plus, être multiples de 8 (la taille est donc $384+i \cdot 8$, *i* entier).

Les mathématiques permettent de démontrer que le décryptage d'un message crypté selon RSA rend bien le message d'origine *à condition que la valeur numérique de ce message soit première avec n*. Un moyen simple d'assurer cela est de ne traiter que des messages de valeur numérique inférieure à **n**, ce qui implique donc un découpage des messages longs en blocs. En fait, *la longueur (exprimée en bytes) du message à crypter n'est pas quelconque* non plus : en fait, il faut que

$$\langle \text{longueur du message} \rangle \leq \langle \text{longueur du module } n \rangle - 2 \cdot \langle \text{longueur du digest} \rangle - 2$$

- le digest dont il est question étant ici inexistant, mais apparaissant dans l'algorithme de signature associé (voir plus loin). Si cette condition n'est pas satisfaite, on obtiendra un message "message too long" et il faudra découper le message original en morceaux plus petits.

- ◆ **ElGamal** : cet algorithme est de la même efficacité que le précédent, en étant un peu plus lent.
- ◆ **LUC** : développé par des chercheurs d'Australie et de Nouvelle-Zélande.

6.4 Du bon usage des chiffrements asymétriques : les clés de session

En fait, *personne n'utilise un chiffrement asymétrique pour coder une série de messages*. En effet, le temps et les ressources nécessaires seraient trop importants. En pratique, on utilise une approche hybride, c'est-à-dire une combinaison de chiffrement asymétrique et symétrique : le système de la **clé de session**. Le mécanisme de base est le suivant.

L'expéditeur qui désire envoyer un message

- ◆ fabrique, d'une manière ou d'une autre, une clé secrète; cette clé est encore appelée la **clé de session**;
- ◆ code le message avec cette clé secrète (chiffrement symétrique);
- ◆ code la clé secrète au moyen de la clé publique du destinataire (chiffrement asymétrique);
- ◆ envoie le message crypté (symétriquement) et la clé secrète cryptée (asymétriquement).

Le destinataire :

- ◆ retrouve la clé secrète en la décryptant au moyen de sa clé privée;
- ◆ utilise la clé secrète obtenue pour décrypter le message.

Cette façon de procéder se retrouve dans tous les domaines de sécurité, comme la protection des messageries électroniques (PGP, S/MIME) ou celle des transactions WEB (SSL).

Une variante, qui évite de transférer la clé de session sur le réseau, consiste à permettre aux deux parties de générer localement la même clé de session sur base de renseignements échangés par le réseau (que celui-ci soit préalablement sécurisé ou non). C'est ce que permet l'algorithme de Diffie-Hellman qui se base sur un échange de nombres premiers et aléatoires. Le danger est cependant qu'un intrus peut se glisser dans cette conversation préalable et se faire passer pour l'expéditeur quand il converse avec le destinataire et vice-versa (attaque "***man in the middle***").

7. Les classes de cryptographie : l'interface et l'implémentation

7.1 L'interface de référence

Le **JCA** (Java Cryptography Architecture) est en fait la spécification formelle des concepts cryptographiques. Plus pragmatiquement, on peut considérer que le JCA fournit des classes et surtout des interfaces que tout le monde est prié d'utiliser; ces éléments se trouvent dans les packages **java.security** et **javax.crypto** (qui font partie intégrante du JDK). Ainsi, par exemple, on trouvera dans **java.security** l'interface **Key**, avec

- ◆ public abstract String getAlgorithm()
- ◆ public abstract byte[] getEncoded()
- ◆ public abstract String getFormat()

tandis que l'on trouvera dans le package **javax.crypto** la classe **Cipher**, avec des méthodes comme

- ◆ public final byte[] crypt(byte[] in)
 - ◆ public final java.lang.String getAlgorithm()
 - ◆ public static Cipher getInstance(java.lang.String algorithm)
- etc

On distingue encore dans les méthodes de ces classes de cryptographie

- ◆ les **APIs** (Application Programming Interface) : il s'agit des méthodes, publiques, que l'on peut appeler directement dans une application;

- ♦ les **SPIs** (Service Provider Interface) : ce sont en fait des méthodes d'interfaces, donc en fait des méthodes sans implémentation, dont le nom est, par convention, préfixé du vocable "engine". Le plus souvent, une API fait usage d'une SPI de manière encapsulée : par exemple, la méthode `initEncrypt(Key)` utilise `engineInitEncrypt(Key)`. Fournir l'implémentation d'une SPI permet donc de se servir de l'API correspondante et l'on disposera en fait d'autant de versions de l'API que l'on fournit de SPIs distinctes.

Le **JCE** (Java Cryptography Extension) est une extension du JCA comportant l'encryptage et l'échange des clés. Il est séparé du JCA de base parce que le gouvernement américain considère que de tels outils sont assimilables à des armes : *il en limite donc l'usage aux USA et au Canada*, mis à part quelques classe de base qui figurent dans le package `java.security`.

7.2 Obtenir les classes : les méthodes factory

Le JCA propose la démarche suivante. Les véritables classes nécessaires au travail cryptographique ne sont pas instanciées au moyen d'un constructeur mais plutôt en utilisant des **méthodes factory**. Ainsi, par exemple, si l'on désire un générateur de clé (objet `KeyGenerator`) utilisant l'algorithme DES, on écrira :

```
KeyGenerator cleGen = KeyGenerator.getInstance("DES");
```

Le générateur de clé, soit l'objet `KeyGenerator`, reste donc une classe conceptuelle, abstraite, qui implémente un interface et à qui l'on peut faire correspondre une version particulière pour l'algorithme précisé (ici DES). Le choix de la classe effectivement utilisée se fait donc dynamiquement, au moment de l'exécution. En travaillant ainsi par interface interposé, on peut écrire un code très général, indépendant de la nature exacte de l'algorithme utilisé, *ce qui ne serait pas le cas avec l'utilisation d'un constructeur*.

7.3 L'implémentation et les providers

Il faudra quand même bien que quelqu'un fournisse les véritables implantations ! Effectivement, Sun a développé de telles classes au sein du JCE (Java Cryptography Extension), extension naturelle du JDK. Le problème, pour rappel, c'est que le gouvernement américain en limite donc l'usage aux USA et au Canada. Que faire alors quand on est Européen (par exemple) ? Plusieurs sociétés ont, fort heureusement, développé des librairies équivalentes au JCE et plus ou moins conformes au JCA. L'usage de ces librairies n'est pas limité géographiquement et, souvent, elles sont même gratuites. Ces sociétés sont encore appelées les **CSP** (Cryptographic Service Provider) ou simplement "providers". Citons :

- ♦ **Cryptix** (<http://www.cryptix.org>), simple librairie exemple en sa version 1.3 (également appelée "**Cryptix JCE**"), compatible avec les JDK 1.5-1.6-1.7 mais restée inchangée depuis 2005; Cryptix est une fondation internationale qui produit des librairies cryptographiques en open-source; l'usage commercial ou non en est gratuit;



- ♦ **Bouncy Castle**, une bibliothèque renommée et très utilisée de cryptographie libre open-source (<http://www.bouncycastle.org>); l'ensemble des contributeurs à cette librairie se désigne par le nom de "**Legion of the Bouncy Castle**" (de fait, ils sont très nombreux !) et constituent un provider pour le JCE et le JCA en fournissant une implémentation du JCE 1.2.1 (avec des classes additionnelles).



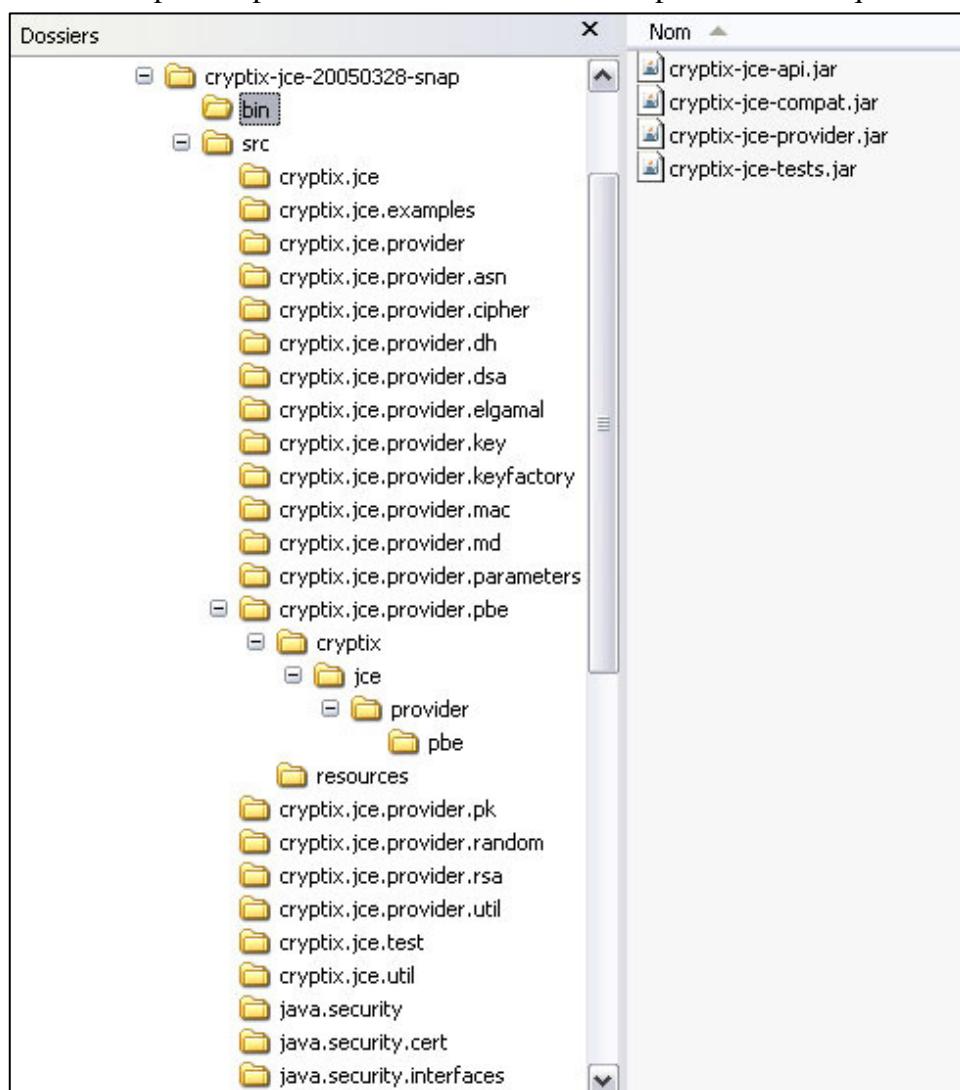
A chacun de ces providers correspond un objet instanciant une classe dérivée de la classe du **JDK Provider**, qui encapsule notamment les notions de nom et de version. Mais surtout, cette classe Provider hérite elle-même de la classe Properties, dont le rôle, comme son nom l'indique, est de contenir une liste de propriétés. Ici, ces propriétés sont les algorithmes effectivement implémentés. On devine sans peine l'usage qui peut être fait de ces listes.

Lorsqu'une méthode factory getInstance(), qui est toujours une méthode de classe, réclame un objet bien précis implémentant un algorithme donné, la classe correspondante va rechercher, par le biais de la classe Security, la liste des providers enregistrés et, selon un ordre de préséance défini par l'utilisateur, va leur demander un objet de la classe algorithm recherché. Si le provider visé la possède, il en fournira une instance. Sinon, la recherche passera au provider suivant ou donnera une exception si la liste est épuisée.

7.4 Des providers complémentaires

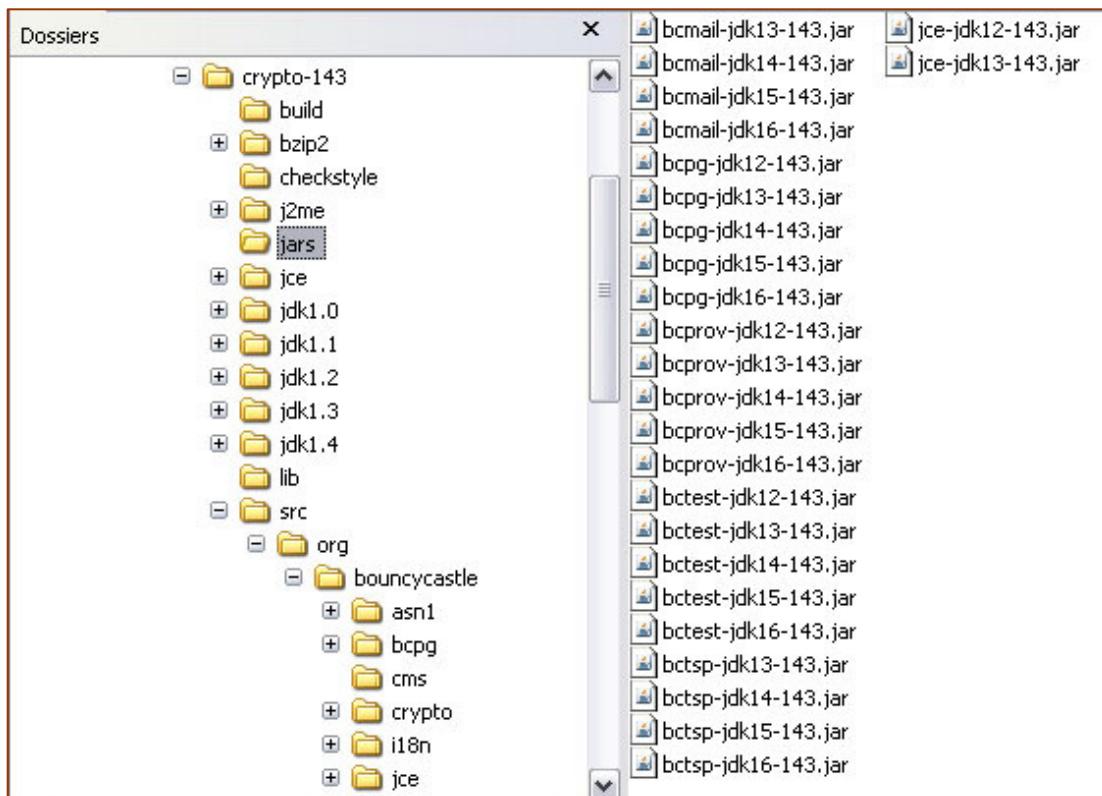
Une liste de providers est fournie par défaut, Sun ... Mais donc libre à nous d'en ajouter d'autres. Nous allons ici utiliser, conjointement, l'implémentation du JCA fournie par la société Cryptix et celle proposée par Bouncy Castle. Dans les deux cas, on peut télécharger les librairies depuis le site Web correspondant.

- a) Dans le cas de Cryptix, on obtient ainsi un fichier du type cryptix-jce-20050328-snap.zip qu'il suffit de décompresser pour obtenir une structure de répertoires classique :



Il suffit de monter au sein de notre projet les 4 fichiers jar à utiliser (cryptix-jce-api.jar, cryptix-jce-compat.jar, cryptix-jce-provider.jar et cryptix-jce-tests.jar) qui contiennent les fichiers *.class pour ainsi avoir mis en place les packages souhaités (le classpath de nos applications non traitées en EDI devra alors contenir en complément c:\cryptix-jce-20050328-snap\bin sous Windows ou /usr/local/cryptix-jce-20050328-snap/bin sous Unix)

b) Pour Bouncy Castle, une fois téléchargé le fichier zip comportant les packages (pour nous, crypto-143.zip), il suffit à nouveau de décompresser ce fichier pour obtenir les jars et outils de la librairie:



On aura compris que le répertoire "jars" contient, sous forme de jars, les classes de la librairie dédicacée à tel ou tel JDK. Donc, dans notre cas, il s'agit principalement de bcprov-jdk16-143.jar et bcmail-jdk16-143.jar. On peut évidemment monter ces jars dans chaque projet qui utilise ces classes, mais il est aussi simple de **placer ces jars dans le répertoire C:\Program Files\Java\jdk1.6.0_10\jre\lib\ext**, ce qui en fait une librairie d'extension définitive du JDK :

Dossiers	Nom	Taille	Type	Date de modification
jdk1.6.0_10	bcmail-jdk16-143.jar	224 Ko	Executable Jar File	15/04/2009 13:59
bin	bcprov-jdk16-143.jar	1.649 Ko	Executable Jar File	15/04/2009 13:59
demo	dnsns.jar	9 Ko	Executable Jar File	16/11/2008 23:51
applets	locatedata.jar	821 Ko	Executable Jar File	16/11/2008 23:52
jfc	meta-index	1 Ko	Fichier	16/11/2008 23:51
jpda	sunjce_provider.jar	167 Ko	Executable Jar File	16/11/2008 23:51
jvmti	sunmscapi.jar	32 Ko	Executable Jar File	16/11/2008 23:51
management	sunpkcs11.jar	220 Ko	Executable Jar File	16/11/2008 23:51
nbproject				
plugin				
scripting				
include				
jre				
bin				
lib				
applet				
audio				
cmm				
deploy				
ext				
fonts				

7.5 Enregistrer un provider

Comment les providers sont-ils enregistrés ? Cela peut se faire dynamiquement en utilisant la méthode

```
public static int addProvider (Provider provider)
```

- par exemple :

```
Security.addProvider(new BouncyCastleProvider());
```

Mais le plus simple, dans le cas où l'on fait systématiquement usage des mêmes providers, est de les définir dans le fichier **java.security** qui se trouve dans le sous-répertoire **/lib/security** du répertoire où se trouve le JDK. Le nouveau provider sera enregistré dans ce fichier **java.security**, après les providers Sun-like : en effet, la première entrée (au moins) est réservée à Sun car c'est lui qui doit être utilisé dans le processus de vérification des jars. En fait, le provider en position 2 vise les clés RSA limitées à 512 bits, ceux à partir de la position 3 supportent des clés allant jusqu'à 2048 bits.

C:\Program Files\Java\jdk1.x.y\jre\lib\security

```
#  
# This is the "master security properties file".  
#  
# In this file, various security properties are set for use by  
# java.security classes. This is where users can statically register  
# Cryptography Package Providers ("providers" for short). The term  
# "provider" refers to a package or set of packages that supply a  
# concrete implementation of a subset of the cryptography aspects of  
# the Java Security API. A provider may, for example, implement one or
```

```

# more digital signature algorithms or message digest algorithms.
#
# Each provider must implement a subclass of the Provider class.
# To register a provider in this master security properties file,
# specify the Provider subclass name and priority in the format
#
#   security.provider.<n>=<className>
#
# This declares a provider, and specifies its preference
# order n. The preference order is the order in which providers are
# searched for requested algorithms (when no specific provider is
# requested). The order is 1-based; 1 is the most preferred, followed
# by 2, and so on.
#
# <className> must specify the subclass of the Provider class whose
# constructor sets the values of various properties that are required
# for the Java Security API to look up the algorithms or other
# facilities implemented by the provider.
#
...
#
# List of providers and their preference orders (see above):
#
security.provider.1=sun.security.provider.Sun
security.provider.2=sun.security.rsa.SunRsaSign
security.provider.3=com.sun.net.ssl.internal.ssl.Provider
security.provider.4=com.sun.crypto.provider.SunJCE
security.provider.5=sun.security.jgss.SunProvider
security.provider.6=com.sun.security.sasl.Provider
security.provider.7=org.jcp.xml.dsig.internal.dom.XMLDSigRI
security.provider.8=sun.security.smartcardio.SunPCSC
security.provider.9=sun.security.msapi.SunMSCAPI
security.provider.10=org.bouncycastle.jce.provider.BouncyCastleProvider
security.provider.11=cryptix.jce.provider.CryptixCrypto

#
# Class to instantiate as the system scope:
#
system.scope=sun.security.provider.IdentityDatabase

```

La syntaxe de définition est fort simple :

security.provider.<numéro d'ordre de préférence>=<nom complet de la classe Provider>

Pour nos nouveaux providers, les classes qui les représentent sont celles citées ci-dessus.

Il est alors assez ais  de passer en revue les providers disponibles et d'afficher leurs listes de propri t s. En effet, la classe Security poss de la m thode de classe :

public static **Provider[] getProviders()**

qui, à partir des données de `java.security`, fournit donc un tableau d'objets instanciant la classe `Provider`, déjà évoquée. Parmi les méthodes de celle-ci, notons :

```
public String getName()
public double getVersion()
```

et aussi, par héritage de la classe `Properties` :

```
public void list(PrintStream out)
```

On peut donc programmer :

CptProviders.java

```
import java.security.*;
public class CptProviders
{
    public static void main(String args[])
    {
        Provider prov[] = Security.getProviders();
        for (int i=0; i<prov.length; i++)
        {
            System.out.println(prov[i].getName() + "/" + prov[i].getVersion());
            prov[i].list(System.out);
        }
    }
}
```

Ce qui peut donner quelque chose du genre :

SUN/1.6

-- listing properties --

```
Alg.Alias.Signature.SHA1/DSA=SHA1withDSA
Alg.Alias.Signature.1.2.840.10040.4.3=SHA1withDSA
Alg.Alias.Signature.DSS=SHA1withDSA
SecureRandom.SHA1PRNG ImplementedIn=Software
KeyStore.JKS=sun.security.provider.JavaKeyStore$JKS
Alg.Alias.MessageDigest.SHA-1=SHA
MessageDigest.SHA=sun.security.provider.SHA
KeyStore.CaseExactJKS=sun.security.provider.JavaKeyStore$Ca...
CertStore.com.sun.security.IndexedCollection ImplementedIn=Software
Alg.Alias.Signature.DSA=SHA1withDSA
KeyFactory.DSA ImplementedIn=Software
KeyStore.JKS ImplementedIn=Software
...
Provider.id className=sun.security.provider.Sun
...
Policy.JavaPolicy=sun.security.provider.PolicySpiFile
Alg.Alias.KeyPairGenerator.1.3.14.3.2.12=DSA
...
```

SunSASL/1.5

-- listing properties --

Provider.id className=com.sun.security.sasl.Provider

...

XMLDSig/1.0

-- listing properties --

SunMSCAPI/1.6

-- listing properties --

...

BC/1.43

-- listing properties --

Alg.Alias.Signature.SHA224withCVC-ECDSA=SHA224WITHCVC-ECDSA

...

Alg.Alias.Cipher.RSA//NOPADDING=RSA

...

Alg.Alias.Mac.HMAC/SHA1=HMACSHA1

...

KeyPairGenerator.ECDSA=org.bouncycastle.jce.provider.asymmet...

...

SecretKeyFactory.PBEWITHSHAAND256BITAES-CBC

Provider.id className=org.bouncycastle.jce.provider.BouncyC...

...

Provider.id name=**BC**

...

CryptixCrypto/1.3

-- listing properties --

Alg.Alias.Mac.HmacSHA0=HMAC-SHA0

...

Alg.Alias.MessageDigest.SHA-0=SHA0

Cipher.DES=cryptix.jce.provider.cipher.DES

...

Provider.id className=cryptix.jce.provider.CryptixCrypto

Alg.Alias.MessageDigest.SHA=SHA1

...

KeyGenerator.TripleDES=cryptix.jce.provider.key.TripleDESKey...

...

Provider.id name=**CryptixCrypto**

...

8. Les limitations cryptographiques des JDK récents

L'utilisation immédiate des jars de Bouncy Castle, par exemple pour générer une clé DES, peut cependant conduire à des problèmes du type suivant :

Génération de clé DES de 128 bits

Exception in thread "main" java.lang.ExceptionInInitializerError

```
at javax.crypto.KeyGenerator.getInstance(DashoA12275)
at cryptobouncycastle.CreationCleSymetrique.creeCleSymetrique
(CreationCleSymetrique.java:37)
```

```
at cryptobouncycastle.CreationCleSymetrique.main(CreationCleSymetrique.java:61)
```

Caused by: **java.lang.SecurityException**: Cannot set up certs for trusted CAs

```
at javax.crypto.SunJCE_b.<clinit>(DashoA12275)
... 3 more
```

Caused by: **java.security.PrivilegedActionException**: java.security.InvalidKeyException:

Public key presented not for certificate signature

```
at java.security.AccessController.doPrivileged(Native Method)
... 4 more
```

Caused by: java.security.InvalidKeyException: Public key presented not for certificate signature

```
at org.bouncycastle.jce.provider.X509CertificateObject.checkSignature(Unknown
Source)
at org.bouncycastle.jce.provider.X509CertificateObject.verify(Unknown Source)
at javax.crypto.SunJCE_b.c(DashoA12275)
at javax.crypto.SunJCE_b.b(DashoA12275)
at javax.crypto.SunJCE_q.run(DashoA12275)
... 5 more
```

Java Result: 1

En fait, le problème concerne les versions 1.4 et supérieures du JDK. Ces JDKs possèdent des limitations cryptographiques dues aux législations de certains pays qui interdisent de chiffrer avec des clés de taille trop importante, ces limites de taille variant selon les algorithmes. On peut régler ce problème en attribuant à notre code la permission CryptoAllPermission. Le mécanisme des permissions est expliqué dans "Java IV: programmation de protocoles applicatifs et de techniques de sécurité" – qu'il nous suffise ici de savoir que l'attribution de permissions ("grant" comme on dit en SQL ;-)) est enregistrée dans des fichiers "policy". On peut télécharger sur le site d'Oracle (Sun) les "**Unlimited Strength Jurisdiction Policy Files**" pour le JDK utilisé (jce_policy6.zip) :

Sun ▾ Java ▾ Solaris ▾ Communities ▾ My SDN Account ▾ Join SDN ▾

Sun Developer Network (SDN)

APIs Downloads Products Support Training Participate

» search tips Search ▾

SDN Home > Java Technology > Java SE >

Java SE Downloads



Download the complete platform and runtime environment
Download the Java SE-JavaFX bundle, and use your creative talents to design a winning application. » Get the bundle

Overview Technologies Documentation Community Support **Downloads**

Latest Release | Next Release (Early Access) | Embedded Use | Real-Time | Previous Releases

Java SE Development Kit (JDK) Bundles

JDK 6 Update 13 with JavaFX SDK For your convenience, Sun has bundled Update 13 of the JDK (the Java development platform) and the JavaFX 1.1 SDK, which provides the JavaFX functionality needed to develop RIAs directly. Each product included is subject to its own license.	Download » ReadMe
JDK 6 Update 13 with Java EE This distribution of the JDK is included in the Java EE 5 SDK, which contains the GlassFish v2.1 application server and provides web services, component-model, management, and communications APIs for	Download

Java Expert?
Get paid everytime you answer Java questions!

Learn More ▾ **LIVEPERSON**

Build and submit a new application **Win up to \$25,000** **FEEDBACK**

latest timezone changes. » Learn more

Java SE 6 Documentation	Download Docs ▾
Java SE 6 JDK Source Code JDK 6 source code is available for those interested in exploring the details of the JDK. This includes schools, universities, companies, and individuals who want to examine the source code for personal interest or research & development. The licensing does not impose restrictions upon those who wish to work on independent open-source projects.	Download
Solaris SPARC Patches	Download
Solaris x86 Patches	Download
Other Downloads Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files 6	Download

Java Training

Related Sites

- » java.com
- » java.net
- » NetBeans ▾
- » Java EE SDK
- » OpenJDK Project
- » Open-Source Java Project

Getting Started?

- » New to Java Center
- » Java Tutorial: Getting Started
- » Tutorials
- » Java SE Training

Sun Net Talk - Java SE 6
Take a tour of the newest

Le jar ainsi obtenu contient principalement deux fichiers jars qui contiennent chacun un fichier policy :

default_US_export.policy (US_export_policy.jar)

```
// Manufacturing policy file.
grant {
    // There is no restriction to any algorithms.
    permission javax.crypto.CryptoAllPermission;
};
```

default_local.policy (local_policy.jar)

```
// Country-specific policy file for countries with no limits on crypto strength.
grant {
    // There is no restriction to any algorithms.
    permission javax.crypto.CryptoAllPermission;
};
```

Il n'y a plus qu'à copier ces deux jars dans le répertoire jre/lib/security/ du JDK : les fichiers policy additionnels tiendront ainsi compagnie au fichier java.policy principal :

Dossiers		Nom	Taille	Type	Date de modification
	jdk1.6.0_10	cacerts	58 Ko	Fichier	16/11/2008 23:52
	jdk1.6.0_10\bin	java.policy	3 Ko	Fichier POLICY	16/11/2008 23:52
	jdk1.6.0_10\demo	java.security	10 Ko	Fichier SECURITY	29/04/2009 8:19
	jdk1.6.0_10\include	java.security~	10 Ko	Fichier SECURITY~	16/11/2008 23:52
	jdk1.6.0_10\jre	javaws.policy	1 Ko	Fichier POLICY	16/11/2008 23:52
	jdk1.6.0_10\jre\bin	local_policy.jar	3 Ko	Executable Jar File	16/11/2008 23:52
	jdk1.6.0_10\jre\lib	US_export_policy.jar	3 Ko	Executable Jar File	16/11/2008 23:52
	jdk1.6.0_10\jre\lib\apple				
	jdk1.6.0_10\jre\lib\audio				
	jdk1.6.0_10\jre\lib\cmm				
	jdk1.6.0_10\jre\lib\deploy				
	jdk1.6.0_10\jre\lib\ext				
	jdk1.6.0_10\jre\lib\fonds				
	jdk1.6.0_10\jre\lib\i386				
	jdk1.6.0_10\jre\lib\im				
	jdk1.6.0_10\jre\lib\images				
	jdk1.6.0_10\jre\lib\management				
	jdk1.6.0_10\jre\lib\security				
	jdk1.6.0_10\jre\lib\servicetag				
	jdk1.6.0_10\jre\lib\zi				

Bon, le décor semble donc cette fois complètement mis en place. Passons donc à l'action ...

9. Un cas pratique d'encryptage : un cryptage symétrique

Comment s'y prendre pour coder une chaîne de caractères en utilisant un algorithme symétrique, disons par exemple DES ?

9.1 Un générateur de clé

Il nous faudra évidemment pour débuter nous procurer une clé de cryptage. Ce sera le travail d'un générateur de clé, objet instanciant une classe de type KeyGenerator. Comme dit plus haut, *nous résistons à la tentation de rechercher un constructeur pour plutôt utiliser une méthode factory* :

```
public static KeyGenerator getInstance(String algorithm, String provider)
    throws java.security.NoSuchAlgorithmException, java.security.NoSuchProviderException
ou
public static KeyGenerator getInstance(String algorithm)
    throws java.security.NoSuchAlgorithmException
```

Ces méthodes réclament donc le nom de l'algorithme pour lequel les clés devront être générées et, dans le premier cas, le nom du provider qui fournira la classe demandée (dans le deuxième cas, l'ordre de préséance des providers dicté dans java.security est d'application). Donc, pour ce qui nous concerne, nous nous procurons donc l'objet désiré par :

```
KeyGenerator cleGen = KeyGenerator.getInstance("DES", "CryptixCrypto");
ou
KeyGenerator cleGen = KeyGenerator.getInstance("DES", "BC");
```

9.2 La génération d'une clé

Pour générer une clé convenable, il faut tout d'abord initialiser le générateur de clés au moyen de la méthode :

```
public void initialize (SecureRandom random)
```

En effet, comme tout générateur de nombres aléatoires, un générateur de clé applique une formule relativement sophistiquée de nature itérative : chaque valeur est calculée à partir de la précédente (ou des deux précédentes). Il faut donc une valeur de départ, imprévisible pour un hacker éventuel. Comme l'heure système (semence classique des générateurs) est vraiment trop facile à découvrir, on utilise des algorithmes plus compliqués pour fournir la génération d'une séquence aléatoire vraiment imprévisible. La classe **SecureRandom**, dérivée de cette bonne vieille Random et se trouvant dans le package java.security, gère ce genre de séquence. Il nous suffit donc en définitive de programmer :

```
cleGen.initialize (new SecureRandom());
```

Obtenir la clé relève du jardin d'enfants, si l'on sait que tout générateur de clé possède la méthode :

```
public SecretKey generateKey()
```

Oui, mais qu'est-ce qu'une **SecretKey** ? Ce n'est jamais qu'un interface qui dérive de l'interface Key (comme d'ailleurs PublicKey et PrivateKey dont nous reparlerons) et qui n'apporte rien de plus que l'algorithme qui lui correspond, sa forme codée utilisable en dehors de la machine virtuelle et le nom de ce format d'encodage. Nous obtiendrons notre clé par :

```
SecretKey cle = cleGen.generateKey();
```

9.3 Obténir un chiffrement

Maintenant que nous disposons d'une clé, il nous faut un objet "outil", dont le rôle est de réaliser le chiffrement d'un message selon l'algorithme DES que nous avons choisi. La classe **Cipher** (définie dans le package javax.crypto) représente un tel chiffrement. La factory à utiliser pour obtenir une instance de cette classe existe en plusieurs versions, dont :

```
public static Cipher getInstance(String algorithm, String provider)
```

Il suffit ici de préciser, dans une chaîne de caractères dont les composantes sont séparées par "/", respectivement le nom de l'algorithme, le mode de chiffrement et le type de padding. Pour notre exemple, cela donnera :

```
Cipher chiffrement = Cipher.getInstance("DES/ECB/PKCS5Padding", "CryptixCrypto");  
ou  
Cipher chiffrement = Cipher.getInstance("DES/ECB/PKCS5Padding", "BC");
```

A nouveau, une version polymorphe ignore le second paramètre, laissant le choix du provider au fichier java.security.

9.4 Le cryptage du message

Maintenant que nous disposons d'un objet chiffrement, il faut d'abord l'initialiser avec la clé obtenue auparavant au moyen de la méthode :

```
public final void init(int opmode, Key key) throws InvalidKeyException
```

le premier paramètre prenant l'une des valeurs :

```
public static final int ENCRYPT_MODE = 1,  
public static final int DECRYPT_MODE = 2,
```

Ensuite, il n'y a plus qu'à réaliser le cryptage en utilisant la méthode (polymorphe d'ailleurs) :

```
public final byte[] doFinal(byte[] in) throws IllegalBlockSizeException
```

Il convient d'observer que *ces méthodes agissent sur un tableau de bytes*, pas sur une chaîne de caractères. En définitive, le petit programme s'écrira donc :

CryptageSymetrique.java

```

package cryptographienewcryptix;

import java.io.*;
import java.security.*;
import javax.crypto.*;

public class CryptageSymetrique
{
    private static String codeProvider = "CryptixCrypto"; //ou "BC";
    public static void main(String args[])
    {
        try
        {
            KeyGenerator cleGen = KeyGenerator.getInstance("DES", codeProvider);
            cleGen.init(new SecureRandom());

            SecretKey cle = cleGen.generateKey();
            System.out.println(" *** Clé générée = " + cle.toString());

            Cipher chiffrement = Cipher.getInstance("DES/ECB/PKCS5Padding",
                codeProvider);
            chiffrement.init(Cipher.ENCRYPT_MODE, cle);
            byte[] texteClair = "Francesca aime James".getBytes();
            byte[] texteCrypté = chiffrement.doFinal(texteClair);
            String texteCryptéAff = new String(texteCrypté);
            System.out.println(new String(texteClair) + " ---> " + texteCryptéAff);
        }
        catch (NoSuchAlgorithmException e)
        { System.out.println("Aie aie " + e.getMessage()); }
        catch(NoSuchProviderException e)
        { System.out.println("Aie aie " + e.getMessage()); }
        catch (KeyException e)
        { System.out.println("Aie aie " + e.getMessage()); }
        catch (Exception e)
        { System.out.println("Aie aie imprévu " + e.getMessage()); }
    }
}

```

On obtient comme résultat, par exemple :

```

*** Clé générée = cryptix.jce.provider.key.RawSecretKey@1bac748
Francesca aime James ---> p°;_6*ßfiñ᷑ ŽÁVa: x\Ñk±ææ

```

9.5 Le décodage

La clé, qui doit rester secrète, peut être sérialisée. On peut donc imaginer que l'expéditeur enregistre la clé générée dans un fichier :

```
ObjectOutputStream cléFich =
new ObjectOutputStream(new FileOutputStream("c:\\java-netbeans-application\\x.ser"));
cléFich.writeObject(cle);
cléFich.close();
```

Il suffit alors que le destinataire possède le fichier de sérialisation (qu'il aura placé dans un répertoire protégé) pour retrouver la clé et décoder un message reçu par le réseau :

```
ObjectInputStream cléFichD =
    new ObjectInputStream(new FileInputStream("c:\\java-netbeans-application\\x.ser"));
cle = (SecretKey) cléFichD.readObject();
cléFichD.close();
System.out.println(" *** Clé récupérée = " + cle.toString());

Cipher chiffrageD = Cipher.getInstance("DES/ECB/PKCS5Padding", "CryptixCrypto");
chiffrageD.init(Cipher.DECRYPT_MODE, cle);
byte[] texteDécodé = chiffrageD.doFinal(texteCrypté);
String texteDécodéClair = new String(texteDécodé);
System.out.println(new String(texteCrypté) + " ---> " + texteDécodéClair);
```

On obtient comme résultat :

| p°,¬6*ßfñň ŽÁVa: ×Ñk±ææ ---> Francesca aime James

9.6 Variante avec Rijndael : un chiffrement symétrique par blocs chaînés

DES utilise un système de blocs indépendants les uns des autres (EBC). Avant de nous résumer avec un exemple réseau, nous pouvons encore évoquer un autre exemple de cryptage symétrique, **Rijndael** (encore connu sous le nom d'AES), qui peut utiliser de plus un système de blocs chaînés (**CBC**). Pour rappel, chaque bloc de texte clair est combiné avec le bloc de texte chiffré précédent, ce qui implique l'existence d'un vecteur d'initialisation (**IV**) afin de permettre le cryptage du premier bloc. C'est cet IV qui marquera la seule différence notable dans la programmation. Il nous faudra donc :

- ◆ produire un second nombre aléatoire qui servira de vecteur d'initialisation; cela peut se faire au moyen de la méthode de SecureRandom :

public void **nextBytes** (byte[] bytes)

la dimension du paramètre déterminant le nombre de bytes à produire;

- ◆ spécifier l'existence de ce vecteur d'initialisation lors de l'initialisation de l'objet de chiffrement en utilisant cette fois la méthode :

```
public final void init (int opmode, Key key, AlgorithmParameterSpec params)
    throws InvalidKeyException, InvalidAlgorithmParameterException
```

La classe IvParameterSpec du package javax.crypto.spec implémente l'interface

AlgorithmParameterSpec, qui n'est qu'un prête-nom puisqu'il s'agit d'un interface vide. La classe proprement dite a pour constructeur :

```
public IvParameterSpec(byte[] iv)
```

et possède la méthode :

```
public byte[] getIV()
```

Le programme suivant crypte puis décrypte un message avec un algorithme AES; il utilise la version 1.3 de Cryptix :

CryptageSymetriqueAES.java

```
package cryptographienewcryptix;

import java.security.*;
import javax.crypto.*;
import javax.crypto.spec.*;

public class CryptageSymetriqueAES
{
    private static String codeProvider = "CryptixCrypto"; //BC";
    public static void main(String args[])
    {
        try
        {
            KeyGenerator cleGen =
                KeyGenerator.getInstance("Rijndael", codeProvider);
            cleGen.init(128, new SecureRandom());
            SecretKey cle = cleGen.generateKey();
            System.out.println(" *** Clé générée = " + cle.toString());

            Cipher chiffrement =
                Cipher.getInstance("Rijndael/CBC/PKCS5Padding", codeProvider);
            byte[] vecteurInit = new byte[16];
            SecureRandom sr = new SecureRandom();
            sr.nextBytes(vecteurInit);
            chiffrement.init(Cipher.ENCRYPT_MODE, cle,
                new IvParameterSpec(vecteurInit));
            byte[] texteClair = "Cherchez la femme : vous trouverez l'homme".getBytes();
            byte[] texteCrypté = chiffrement.doFinal(texteClair);
            String texteCryptéAff = new String (texteCrypté);
            System.out.println(new String(texteClair) + " ---> " + texteCryptéAff);
            chiffrement.init(Cipher.DECRYPT_MODE, cle,
                new IvParameterSpec(vecteurInit));
            byte[] texteDécodé = chiffrement.doFinal(texteCrypté);
            String texteDécodéClair = new String (texteDécodé);
            System.out.println(new String(texteCrypté) + " ---> " + texteDécodéClair);
        }
    }
}
```

```
        catch (NoSuchAlgorithmException e)
        { System.out.println("Aie aie " + e.getMessage()); }
        catch(NoSuchProviderException e)
        { System.out.println("Aie aie " + e.getMessage()); }
        catch (KeyException e)
        { System.out.println("Aie aie " + e.getMessage()); }
        catch (Exception e)
        { System.out.println("Aie aie imprévu " + e.getMessage()); }
    }
```

Ce qui donne :

*** Clé générée = cryptix.jce.provider.key.RawSecretKey@9fef6f
Cherchez la femme : vous trouverez l'homme -->
á/e2«[—] ñÍD2‡õâ"¿zpc?Ù{Ýp`H† 2çü↑ ˜ŒâOq4k?uxaÇ(yô-
»á/e2«[—] ñÍD2‡õâ"¿zpc?Ù{Ýp`H† 2çü↑ ˜ŒâOq4k?uxaÇ(yô- --> Cherchez la femme : vous
trouverez l'homme

9.7 L'envoi d'un message crypté par le réseau

Pour nous résumer, nous allons à présent envisager le cas concret suivant. D'un côté, nous avons une machine principale (appelons-la "le serveur", même si elle ne rend ici aucun service); c'est la base de nos agents secrets. Pour nos essais, cette machine s'appelle "claude" et utilise le port 50000. Au préalable, on y génère une clé secrète pour DES, qui est sérialisée dans un fichier x.ser, comme expliqué ci-dessus. Ce serveur se met en attente de la réception d'un message crypté selon cette clé.

De l'autre côté, nous avons une machine utilisée par l'un de nos agents secrets (appelons-la "le client"). Il y dispose du fichier x.ser. Il va envoyer un message crypté avec la clé vers le centre. Ce client s'écrira :

```
ClientCryptoSym.java
package cryptographienewcryptix;

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

import java.security.*;
import javax.crypto.*;

public class ClientCryptoSym
{
    private static String codeProvider = "CryptixCrypto"; // "BC";
    public static void main(String args[])
    {
        Socket cliSock = null;
        DataOutputStream dos=null;
        String ligneDuServeur;
```

```

try
{
    cliSock = new Socket("claude", 50000);
    System.out.println(cliSock.getInetAddress().toString());
    dos = new DataOutputStream(cliSock.getOutputStream());
}
catch (UnknownHostException e)
{
    System.err.println("Erreur ! Host non trouvé [" + e + "]");
}
catch (IOException e)
{
    System.err.println("Erreur ! Pas de connexion ? [" + e + "]");
}
if (cliSock==null || dos==null) System.exit(1);

try
{
    int c;
    byte b;
    System.out.println("Message à envoyer au serveur : ");
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    String msgClient = in.readLine();

    System.out.println("Récupération de la clé");
    ObjectInputStream cléFichD = new ObjectInputStream(
        new FileInputStream("c:\\java-netbeans-application\\x.ser"));
    SecretKey cle = (SecretKey) cléFichD.readObject();
    cléFichD.close();
    System.out.println(" *** Clé récupérée = " + cle.toString());

    System.out.println("Cryptage du message");
    Cipher chiffrement = Cipher.getInstance("DES/ECB/PKCS5Padding",
        codeProvider);
    chiffrement.init(Cipher.ENCRYPT_MODE,cle);
    byte[] texteClair = msgClient.toString().getBytes();
    byte[] texteCrypté = chiffrement.doFinal(texteClair);
    String texteCryptéAff = new String (texteCrypté);
    System.out.println(new String(texteClair) + " ---> " + texteCryptéAff);

    System.out.println("Envoi du message crypté");
    dos.write(texteCrypté);
    dos.flush();
    dos.close(); cliSock.close();
    System.out.println("Client déconnecté");
}
catch (UnknownHostException e)
{ System.err.println("Erreur ! Host non trouvé [" + e + "]"); }
catch (IOException e)
{ System.err.println("Erreur ! Pas de connexion ? [" + e + "]"); }

```

```

        catch (NoSuchAlgorithmException e)
        { System.out.println("Aie aie " + e.getMessage()); }
        catch(NoSuchProviderException e)
        { System.out.println("Aie aie " + e.getMessage()); }
        catch (KeyException e)
        { System.out.println("Aie aie " + e.getMessage()); }
        catch (Exception e)
        { System.out.println("Aie aie imprévu " + e.getMessage() + e.getClass()); }
    }
}

```

Le serveur va donc voir arriver par le réseau un ensemble de bytes qui constituent le message codé. Il le décryptera au moyen de la même clé secrète.

Une petite difficulté se fait cependant jour : le serveur ne peut savoir combien de bytes sont envoyés par le client et il ne peut être question de guetter un caractère terminateur (comme \n), puisque le hasard du cryptage pourrait précisément générer un tel caractère ! Il faudra donc lire byte par byte sur le flux fourni par la socket et placer les bytes recueillis dans un tableau de bytes. Le problème, c'est qu'il faudrait dimensionner ce tableau au préalable ...On utilisera en fait ici un flux instance de la classe **ByteArrayOutputStream**. Un tel flux représente une succession de bytes en mémoire, l'intérêt étant que la quantité de bytes qu'il peut recevoir est indéterminée. On récupère les bytes proprement dits au moyen de la méthode :

```
public synchronized byte[] toByteArray()
```

Notre serveur s'écrira donc en définitive :

ServeurCryptoSym.java

```

package cryptographienewcryptix;

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

import java.security.*;
import javax.crypto.*;

public class ServeurCryptoSym
{
    private static String codeProvider = "BC"; //CryptixCrypto";
    public static void main(String args[])
    {
        ServerSocket serSock = null;
        try
        {
            serSock = new ServerSocket(50000);
        }
        catch (IOException e)
        { System.err.println("Erreur de port d'écoute ! ? [" + e + "]"); System.exit(1); }
    }
}

```

```

Socket serSockCli = null;
System.out.println("Serveur en attente");
try
{
    serSockCli = serSock.accept();
}
catch (IOException e)
{
    System.err.println("Erreur d'accept ! ? [" + e + "]");
    System.exit(1);
}

byte b;
DataInputStream dis=null;
ByteArrayOutputStream baos = new ByteArrayOutputStream();

byte[] texteCrypté;
try
{
    dis = new DataInputStream( new BufferedInputStream(
        serSockCli.getInputStream()));
    if (dis==null) System.exit(1);

    while (true)
    {
        b = dis.readByte();
        baos.write(b);
    }
}
catch (EOFException e)      // Mouais ... OK parce que l'on ferme la connexion ...
{
    texteCrypté = baos.toByteArray();
    System.out.println("Msg reçu = " + new String(texteCrypté));
    // Déchiffrement
    try
    {
        ObjectInputStream cléFichD = new ObjectInputStream(
            new FileInputStream("c:\\java-netbeans-application\\x.ser"));
        SecretKey cle = (SecretKey) cléFichD.readObject();
        cléFichD.close();
        System.out.println(" *** Clé récupérée = " + cle.toString());
    }

    Cipher chiffrementD =
        Cipher.getInstance("DES/ECB/PKCS5Padding", codeProvider);
    chiffrementD.init(Cipher.DECRYPT_MODE,cle);
    byte[] texteDécodé = chiffrementD.doFinal(texteCrypté);
    String texteDécodéClair = new String (texteDécodé);
    System.out.println(new String(texteCrypté) + " ---> " + texteDécodéClair);
}
catch (NoSuchAlgorithmException ee)

```

```
{ System.out.println("Aie aie " + ee.getMessage()); }
catch(NoSuchProviderException ee)
{ System.out.println("Aie aie " + ee.getMessage()); }
catch (KeyException ee)
{ System.out.println("Aie aie " + ee.getMessage()); }
catch (FileNotFoundException ee)
{ System.out.println("Aie aie " + ee.getMessage()); }
catch (Exception ee)
{ System.out.println("Aie aie imprévu " + ee.getMessage() + ee.getClass()); }
}
catch (IOException e)
{ System.err.println("Erreur ! ? [" + e + "]"); }
finally
{
    try
    {
        dis.close();
        serSockCli.close();
        serSock.close();
        System.out.println("Serveur déconnecté");
    }
    catch (IOException e)
    { System.err.println("Erreur ! ? [" + e + "]"); }
}
}
}
```

Deux exécutions successives peuvent donner :

```
claude/192.168.2.2
Message à envoyer au serveur :
Francesca aime James
Récupération de la clé
Cryptage du message
Francesca aime James ---> *Nïë\u001bbu?]Üvè`@¬8®Zäÿ?#GG
Envoi du message crypté
Client déconnecté
```

```
Serveur en attente
Msg reçu = *Nïë\u001bbu?]Üvè`@¬8®Zäÿ?#GG
*** Clé récupérée = cryptix.provider.key.RawSecretKey@5cf3aacc
*Nïë\u001bbu?]Üvè`@¬8®Zäÿ?#GG ---> Francesca aime James
Serveur déconnecté
```

et

```
claude/192.168.2.2
Message à envoyer au serveur :
Le petit cochon est dans le pré
Récupération de la clé
```

Cryptage du message

Le petit cochon est dans le pré ---> µñÅ\bl½[ÄNL/?ì;?ää?U-xFCöEÖéx:&p?ù

Envoi du message crypté

Client déconnecté

Serveur en attente

Msg reçu = µñÅ\bl½[ÄNL/?ì;?ää?U-xFCöEÖéx:&p?ù

*** Clé récupérée = cryptix.provider.key.RawSecretKey@5cf3aacc

µñÅ\bl½[ÄNL/?ì;?ää?U-xFCöEÖéx:&p?ù ---> Le petit cochon est dans le pré

Serveur déconnecté

10. Un deuxième cas pratique : un cryptage asymétrique

10.1 L'obtention d'une paire de clés

Tenant compte des remarques déjà formulées plus haut concernant la faiblesse d'une système à clé secrète unique, on pourrait s'attendre à la reprise de l'exemple ci-dessus en utilisant une solution à chiffrement asymétrique, autrement dit vers solution clé publique/clé privée.

Cependant, pour rappel, la pratique le plus courante consiste plutôt à utiliser de manière conjointe les deux mécanismes de clé (clé secrète d'une part, couple clé publique/clé privée). Pour rappel (voir p.201), l'idée est alors la suivante :

- ◆ l'expéditeur génère une clé secrète, donc de chiffrement symétrique;
- ◆ il crypte le message à envoyer avec cette clé;
- ◆ il crypte cette clé secrète au moyen d'un chiffrement asymétrique initialisé avec la clé publique du destinataire;
- ◆ ces deux messages (message de base et message de clé secrète) cryptés sont envoyés au destinataire;
- ◆ celui-ci décrypte le message de clé secrète reçu au moyen d'un chiffrement asymétrique initialisé avec sa clé privée;
- ◆ ayant ainsi récupéré la clé secrète, il peut déchiffrer le message de base avec un chiffrement symétrique utilisant cette clé secrète; cette clé qui sera utilisée pour la suite de la communication est souvent appelée la **clé de session**.

Nous allons reparler des couples de clés ci-dessous. Pour l'instant, voyons comment générer un tel couple. En fait, tout se passe d'une manière très similaire à celle de la génération d'une clé secrète. Nous allons obtenir un objet **KeyPairGenerator**, classe se trouvant dans le package java.security. Nous appellerons une des méthodes factory de KeyPairGenerator :

```
public static KeyPairGenerator getInstance(String algorithm)
    throws NoSuchAlgorithmException
public static KeyPairGenerator getInstance(String algorithm, String provider)
    throws NoSuchAlgorithmException
```

On peut donc se contenter de programmer :

```
KeyPairGenerator RSAgenCles = KeyPairGenerator.getInstance("RSA");
```

sans préciser de provider – le premier provider trouvé sera Sun :

```
*** Cle publique generee = Sun RSA public key, 512 bits  
*** Cle privee generee = Sun RSA private CRT key, 512 bits
```

On peut aussi préciser un provider :

```
KeyPairGenerator RSAgenCles = KeyPairGenerator.getInstance("RSA", "CryptixCrypto");
```

ce qui donnera des clés instances de classe de ce provider :

```
*** Cle publique generee = cryptix.jce.provider.rsa.RSAPublicKeyCryptix@10dd1f7  
*** Cle privee generee = cryptix.jce.provider.rsa.RSAPrivateCrtKeyCryptix@53c015
```

L'initialisation peut se réaliser en utilisant :

```
public void initialize(int keySize, java.security.SecureRandom source)
```

où le premier paramètre représente le nombre de bits du module (une version polymorphe permet de préciser l'exposant). Pour nous, cela pourrait donner :

```
RSAgenCles.initialize(1024, new SecureRandom());
```

La méthode :

```
public java.security.KeyPair generateKeyPair()
```

va nous fournir le couple de clés souhaité sous forme d'un objet KeyPair (du package java.security). Celui-ci n'est jamais qu'un aggrégat de deux composantes que l'on obtient au moyen des méthodes :

```
public PublicKey getPublic()  
public PrivateKey getPrivate()
```

où **PublicKey** et **PrivateKey** sont à nouveau des interfaces qui dérivent de l'interface Key.

ClesPourCryptageAsymétrique.java

```
package cryptographienewcryptix;  
  
import java.io.*;  
import java.security.*;  
  
public class ClesPourCryptageAsymétrique  
{  
    private static String codeProvider = "CryptixCrypto"; // "BC";  
    private static final SecureRandom prng = new SecureRandom();  
    public static void main(String args[])  
    {
```

```

try
{
    // Génération des clés
    System.out.println("Tentative d'obtention d'un generateur de cle");
    KeyPairGenerator genCles = KeyPairGenerator.getInstance("RSA");
    //, codeProvider);
    System.out.println("Tentative d'initialisation du generateur de cle");
    int se = 512; // par exemple
    genCles.initialize(se, prng);
    System.out.println("Tentative d'obtention de cles");
    KeyPair deuxCles = genCles.generateKeyPair();
    PublicKey cléPublique = deuxCles.getPublic();
    PrivateKey cléPrivée = deuxCles.getPrivate();
    System.out.println(" *** Cle publique generee = " + cléPublique);
    System.out.println(" *** Cle privee generee = " + cléPrivée);
    // Sérialisation de clés
    System.out.println(" *** Cle publique generee serialisee");
    ObjectOutputStream cléPubliqueFich = new ObjectOutputStream(
        new FileOutputStream("c:\\java-netbeans-application\\cles\\xp.ser"));
    System.out.println("fichier ouvert");
    cléPubliqueFich.writeObject(cléPublique);
    System.out.println("cle ecrite");
    cléPubliqueFich.close(); System.out.println("fichier ferme");
    System.out.println(" *** Cle privee generee serialisee");
    ObjectOutputStream cléPrivéeFich = new ObjectOutputStream(
        new FileOutputStream("c:\\java-netbeans-application\\cles\\xs.ser"));
    cléPrivéeFich.writeObject(cléPrivée);
    cléPrivéeFich.close();
    System.out.println("** Opérations terminées **"); System.exit(0);
}
catch (NoSuchAlgorithmException e)
{
    System.out.println("Aie aie " + e.getMessage());
}
catch (FileNotFoundException e)
{
    System.out.println("Aie aie fichier non trouvé " + e.getMessage());
}
catch (Exception e)
{
    System.out.println("Aie aie imprévu ou " + e.getMessage() + " -- " + e.getClass());
}
}
}

```

On obtient ainsi, par exemple, sans précision de provider (donc en utilisant Sun vu notre fichier java.security) :

Tentative d'obtention d'un generateur de cle
 Tentative d'initialisation du generateur de cle
 Tentative d'obtention de cles
 *** Cle publique generee = Sun RSA public key, 512 bits
modulus:
 872707592415633773198252549095848396306716123375344946025001468060400293844
 502833017212489338879759991374551726717039291272057377210184294130243432850
 2297
public exponent: 65537

*** **Cle privee generee** = Sun RSA private CRT key, 512 bits

modulus:

872707592415633773198252549095848396306716123375344946025001468060400293844
502833017212489338879759991374551726717039291272057377210184294130243432850
2297

public exponent: 65537

private exponent:

131205088851659971730814385251711159326946213034122308820732402532906970036
005617184215509256686573679606613266787081084146852291357885903355053788848
2257

prime p:

108979551038682381869033195391796508904459113913385448753199563777492168622
373

prime q:

800799401445382621117807238559904574185441593752806620051252123162136916507
89

prime exponent p:

129537620365798824291586217266901872890998443228294344536280971333241374119
7

prime exponent q:

498536942169638691755901785758336613314097639884561546883303883684257536865
crt coefficient:

678371138071741660714167884247416078610974925894190956321346531524526730069
29

*** Cle publique generee serialisee

fichier ouvert

cle ecrite

fichier ferme

*** Cle privee generee serialisee

** Opérations terminées **

Que de renseignements ☺ ... Par contre, si on utilise explicitement l'antique provider Cryptix, l'affichage est moins prolix ☺ :

Tentative d'obtention d'un generateur de cle

Tentative d'initialisation du generateur de cle

Tentative d'obtention de cles

*** **Cle publique generee** = cryptix.jce.provider.rsa.RSAPublicKeyCryptix@e2eec8

*** **Cle privee generee** = cryptix.jce.provider.rsa.RSAPrivateCrtKeyCryptix@aa9835

*** Cle publique generee serialisee

fichier ouvert

cle ecrite

fichier ferme

*** Cle privee generee serialisee

** Opérations terminées **

A titre d'information, on peut cependant obtenir des informations supplémentaires si l'on considère que les clés générées sont en fait des implémentations des interfaces

RSAPublicKey et **RSAPrivateKey** qui héritent

- ♦ respectivement de PublicKey et de PrivateKey, qui apportent les méthodes

```
public java.math.BigInteger getPublicExponent()
public java.math.BigInteger getPrivateExponent
```

- ♦ conjointement de **RSAKey** qui apporte la méthode :

```
public java.math.BigInteger getModulus()
```

On remarquera que tous les renseignements sont fournis sous forme d'objets qui sont des instances de la classe **BigInteger** (définie dans le package `java.math`). Avec un nom pareil, son rôle est clair ...

Si donc nous ajoutons dans le code précédent :

```
System.out.println(" *** Cle publique generee = " + cléPublique);
// Si nécessaire
System.out.println("Module = " + ((RSAPublicKey)cléPublique).getModulus().toString());
System.out.println("Exposant public = " +
    ((RSAPublicKey)cléPublique).getPublicExponent().toString());
System.out.println(" *** Cle privee generee = " + cléPrivée);
// Si nécessaire
System.out.println("Module = " + ((RSAPrivateKey)cléPrivée).getModulus().toString());
System.out.println("Exposant privé = " +
    ((RSAPrivateKey)cléPrivée).getPrivateExponent().toString());
```

nous obtiendrons :

```
Tentative d'obtention d'un generateur de cle
Tentative d'initialisation du generateur de cle
Tentative d'obtention de cles
*** Cle publique generee = cryptix.jce.provider.rsa.RSAPublicKeyCryptix@e2eec8
Module =
755980085666032994657146986774516167886464694884804968419977202235503455262
298123717264001446113941813788771344196697646175073602708163958665422196372
3101
Exposant public = 65537
*** Cle privee generee = cryptix.jce.provider.rsa.RSAPrivateCrtKeyCryptix@aa9835
Module =
755980085666032994657146986774516167886464694884804968419977202235503455262
298123717264001446113941813788771344196697646175073602708163958665422196372
3101
Exposant privé =
673307560619594212401203436501953228245616128910784228858111742900748228995
213290024078699247778336699668022337028565495245314201406878234238170979555
0593
*** Cle publique generee serialisee
fichier ouvert
cle ecrite
```

```
fichier ferme  
*** Cle privee generee serialisee  
** Opérations terminées **
```

A remarquer que si le provider est Bouncy Castle, on obtient les renseignements immédiatement, comme pour Sun.

10.2 Un cryptage-décryptage élémentaire

Bien sûr, une fois les clés générées, il reste à les sérialiser puis à distribuer la clé publique (mais ce n'est pas si simple : voir les certificats). Limitons-nous ici à vérifier le cryptage-décryptage au moyen d'une telle paire de clés. Ceci ressemble fort à ce qui se programme pour un cryptage-décryptage symétrique, si ce n'est que :

- ♦ l'objet chiffrement s'obtient cette fois avec :

```
Cipher chiffrement= Cipher.getInstance("RSA/ECB/PKCS#1","CryptixCrypto");
```

- un simple paramètre "RSA" impliquerait qu'aucun padding ne serait utilisé, ce qui provoquerait une exception dans presque tous les cas;

- ♦ l'encryptage s'initialise avec la clé publique, le décryptage avec la clé privée.

CryptageAsymétrique.java

```
package cryptographienewcryptix;  
  
import java.security.*;  
import javax.crypto.*;  
import java.io.*;  
  
public class CryptageAsymetrique  
{  
    private static String codeProvider = "CryptixCrypto";  
    public static void main(String[] args)  
    {  
        try  
        {  
            ObjectInputStream cléPubliqueFich = new ObjectInputStream(  
                new FileInputStream("c:\\java-netbeans-application\\cles\\xp.ser"));  
            PublicKey cléPublique = (PublicKey)cléPubliqueFich.readObject();  
            cléPubliqueFich.close();  
  
            ObjectInputStream cléPrivéeFich = new ObjectInputStream(  
                new FileInputStream("c:\\java-netbeans-application\\cles\\xs.ser"));  
            PrivateKey cléPrivee = (PrivateKey)cléPrivéeFich.readObject();  
            cléPrivéeFich.close();  
        }  
    }  
}
```

```
/* cryptage */
System.out.println("Tentative d'obtention d'un objet de chiffrement");
byte[] texteClair="Le petit cochon est dans la prairie".getBytes();
int longueur = texteClair.length;
Cipher chiffrement= Cipher.getInstance("RSA/ECB/PKCS#1",codeProvider);
chiffrement.init(Cipher.ENCRYPT_MODE, cléPublique);
byte[] texteCrypté = chiffrement.doFinal (texteClair);
System.out.println(new String(texteClair) + " ---> " + texteCrypté);

/* décryptage */
Cipher chiffrementd=
    Cipher.getInstance("RSA/ECB/PKCS#1",codeProvider);
chiffrementd.init(Cipher.DECRYPT_MODE, cléPrivee);
byte[] texteClair2 = chiffrementd.doFinal (texteCrypté);
System.out.println(texteCrypté + " ---> " +
    new String(texteClair2).substring(0,longueur));
}
catch (Exception e)
{ System.out.println("Aie aie imprévu " + e.getMessage() + e.getClass()); }
}
}
```

Une exécution à la console donne :

```
Tentative d'obtention d'un generateur de cle
Tentative d'initialisation du generateur de cle
Tentative d'obtention de cles
Le petit cochon est dans la prairie ---> [B@15cda3f
[B@15cda3f ---> Le petit cochon est dans la prairie
```

Donc, cela fonctionne vraiment ;-)

Remarque

Si on utilise le provider Bouncy Castle, l'expression de l'algorithme de padding est légèrement différente ☺ :

```
Cipher chiffrement= Cipher.getInstance("RSA/ECB/PKCS1Padding",codeProvider);
Cipher chiffrementd= Cipher.getInstance("RSA/ECB/PKCS1Padding",codeProvider);
```

11. Les message digests

11.1 Les fonctions de hachage

Le problème de l'**intégrité** des données transmises est souvent sous-estimé, en tous cas moins considéré que celui de la confidentialité ou de l'authentification. Pourtant, il importe de savoir si les données que l'on obtient par le réseau (par exemple) sont restées ce qu'elles étaient à leur envoi.

Dans ce contexte, on utilise un **message digest** du document à envoyer : il s'agit en fait de la "valeur de hachage" de celui-ci. Celle-ci s'obtient en utilisant une **fonction de hachage** qui, recevant un ensemble des données d'une taille quelconque, fournit une chaîne de taille fixe (typiquement, 128 bits). Comme c'est l'ensemble des données qui est utilisé, la probabilité de trouver deux fois la même valeur résultante pour des messages différents est extrêmement faible. De plus, les fonctions de hachage idéales ne peuvent être inversées (ou, du moins, il faut un temps quasiment infini pour y parvenir) : il est donc impossible de retrouver la donnée qui a produit un digest.

L'intégrité est donc vérifiée

- ◆ en calculant le message digest sur le texte obtenu;
- ◆ en le comparant au message digest qui accompagnait le texte.

En cas d'égalité, on peut penser que le texte original n'a pas été modifié. Sauf si un hacker a capté le message, recomposé à sa sauce et généré un nouveau digest ! Ceci ne peut être évité que si on utilise une signature digitale (voir plus loin).

11.2 Un exemple

Supposons à nouveau que le message à transmettre par James à Francesca soit simplement "hello". L'expéditeur (James) et le destinataire (Francesca) se mettent d'accord sur une méthode de hachage, disons par exemple :

$$h(\text{message}) = (\sum \text{(codes ASCII des caractères)}) \% 67$$

message digest d'origine :

message	"hello"	0110 1000	0110 0101	0110 1100	0110 1100	0110 1111
valeur de hachage = $h1 = (104+101+108+108+111) \% 67 = 63 = d$						

concaténation

message avec digest	$d\text{"hello"}$	63	104	101	108	108	111
---------------------	-------------------	----	-----	-----	-----	-----	-----



(H=Hachage)

message digest à l'arrivée :

message avec digest	$d\text{"hello"}$	63	104	101	108	108	111
$h1 = 63$							
$h2 = \text{valeur de hachage} = (104+101+108+108+111) \% 67 = 63$							

conclusion

$h1 = h2 \rightarrow$ le message n'a pas été modifié

11.3 Quelques algorithmes de message digest courants

On peut citer :

- ◆ **MD2, MD4 et MD5** (R. Rivest) : ils produisent des digests de 128 bits; le premier fut optimisé pour les machines 8 bits, les deux autres pour celles à 32 bits. En fait, MD4 n'est pas très sûr.
- ◆ **SHA-1** (Secure Hash Algorithm) : un peu plus sûr que MD5, il a été adopté comme standard par le NIST sous le nom de **SHS** (Secure Hash Standard). Le digest produit est de 160 bits pour des blocs d'entrée de 512 bits (sauf le dernier, qui ne comprend que 448 bits, car l'algorithme lui ajoute le codage sur 64 bits de la longueur effective de l'entrée).
- ◆ **RIPEMD-160** : il produit des digests de 160 bits.

11.4 Utilisation des digests pour les mots de passe

Classiquement, un client s'authentifie auprès d'un serveur par son nom et un mot de passe. Le problème est que l'on souhaite ne pas laisser ce mot de passe circuler en clair sur le réseau. Le principe d'une solution est assez simple.

1) Le client :

- ◆ connaît évidemment à l'avance son mot de passe pour accéder au serveur visé;
- ◆ crée un digest avec son mot de passe;
- ◆ envoie son nom et le digest au lieu de son mot de passe en clair.

Le serveur :

- ◆ reçoit le nom et le digest;
- ◆ recherche le nom dans son fichier (ou sa base de données) et calcule le digest correspondant;
- ◆ si le digest calculé correspond au digest reçu, l'utilisateur est authentifié.

2) Evidemment, un hacker peut se mettre à l'écoute et utiliser le digest capté dans une communication ultérieure. Aussi, le digest est plutôt calculé avec le mot de passe complété d'un nombre aléatoire et de l'heure. Ces deux dernières informations seront envoyées en clair au serveur, afin qu'il puisse lui aussi calculer le digest pour le comparer à celui qu'il a reçu.

3) On peut encore renforcer la méthode en recalculant un deuxième digest à partir du premier, ou encore en utilisant un second nombre aléatoire et l'heure de cette deuxième génération.

La seule limite de cette méthode est qu'elle ne prémunit pas contre un hacker particulièrement patient qui tenterait des mots de passe successifs (*dictionary attack*) – mais cela lui prendrait évidemment un temps très long.

11.5 Un exemple de login avec mot de passe

Le client va adopter la deuxième stratégie pour contacter un serveur qui est sensé le reconnaître sur foi de son mot de passe. Il va donc falloir :

1) obtenir une instance de digest, qui utilise l'algorithme SHA-1 et qui est fournie par le provider Cryptix, au moyen de la méthode factory :

```
public static MessageDigest getInstance(String algorithm, String provider)  
throws NoSuchAlgorithmException, NoSuchProviderException
```

Donc, dans notre cas :

MessageDigest md = MessageDigest.getInstance("SHA-1", "Cryptix");

2) préparer le digest à proprement parler en y plaçant les "ingrédients" (nom, mot de passe, nombre aléatoire et heure) au moyen de la méthode

public void update(byte input[])

qui permet d'ajouter au futur digest le tableau de bytes passé comme paramètre. Pour ce qui est des nom et mot de passe, qui sont des instances de String, il suffira d'utiliser la méthode **getBytes**. Par contre, pour le nombre aléatoire et l'heure, le plus simple est d'utiliser une instance de la classe **ByteArrayOutputStream**, qui représente un tableau de bytes sous la forme d'un flux ! Il suffit de construire dessus un classique **DataOutputStream** pour pouvoir utiliser les habituelles méthodes **writeXXX**.

3) enfin, appeler la méthode

public byte[] digest(byte input[])

qui applique effectivement l'algorithme sélectionné pour fournir le digest sous forme de bytes.

ClientPassword.java

```
package cryptographienewcryptix;

import java.io.*;
import java.net.*;
import java.util.Date;
import java.security.*;

public class ClientPassword
{
    private static String codeProvider = "BC"; //CryptixCrypto";
    public static void main(String args[])
    {
        Socket cliSock = null;
        DataOutputStream dos=null;
        DataInputStream dis=null;

        String ligneDuServeur;

        try
        {
            cliSock = new Socket("claude", 50000);
            System.out.println(cliSock.getInetAddress().toString());

            dos = new DataOutputStream(cliSock.getOutputStream());
            dis = new DataInputStream(cliSock.getInputStream());
        }
```

```

        catch (UnknownHostException e)
        {
            System.err.println("Erreur ! Host non trouvé [" + e + "]");
        }
        catch (IOException e)
        {
            System.err.println("Erreur ! Pas de connexion ? [" + e + "]");
        }

        if (cliSock==null || dis==null || dos==null) System.exit(1);

        try
        {
            int c;
            byte b;
            StringBuffer msgClient = new StringBuffer();

            System.out.println("Message à envoyer au serveur : ");
            String user = "Claude", password = "beaugosse";

            System.out.println("Instanciation du message digest");
            MessageDigest md = MessageDigest.getInstance("SHA-1", codeProvider);
            md.update(user.getBytes());
            md.update(password.getBytes());

            long temps = (new Date()).getTime();
            double alea = Math.random();
            ByteArrayOutputStream baos = new ByteArrayOutputStream();
            DataOutputStream bdos = new DataOutputStream(baos);
            bdos.writeLong(temps); bdos.writeDouble(alea);

            md.update(baos.toByteArray());
            byte[] msgD = md.digest();

            System.out.println("Envoi du message digest");
            dos.writeUTF(user);
            dos.writeLong(temps);
            dos.writeDouble(alea);
            dos.writeInt(msgD.length);
            dos.write(msgD);

            String réponse = dis.readUTF();
            System.out.println("Réponse du serveur = " + réponse);
            dos.close();dis.close();
            cliSock.close();
            System.out.println("Client déconnecté");
        }
        catch (UnknownHostException e)
        { System.err.println("Erreur ! Host non trouvé [" + e + "]"); }
        catch (IOException e)
    
```

```
{ System.err.println("Erreur ! Pas de connexion ? [" + e + "]"); }
catch (NoSuchAlgorithmException e)
{ System.out.println("Aie aie " + e.getMessage()); }
catch(NoSuchProviderException e)
{ System.out.println("Aie aie " + e.getMessage()); }
catch (Exception e)
{ System.out.println("Aie aie imprévu " + e.getMessage() + e.getClass()); }
}
```

Le serveur reçoit le digest, ainsi que les éléments autres que le mot de passe en clair. Il va recomposer un nouveau digest et le comparer à celui qu'il a reçu par :

```
public static boolean isEqual(byte digesta[], byte digestb[])
```

En fait, cette méthode de classe se contente de comparer les deux digests byte à byte. Evidemment, en pratique, le serveur recherchera le client dans une base de données pour y retrouver le mot de passe enregistré. Ici, pour simplifier, la méthode *chercheMotDePasse()* renvoie une chaîne exacte ou pas (⊗ triste ...) !

ServeurPassword.java

```
package cryptographienewcryptix;

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

public class ServeurPassword
{
    private static String codeProvider = "BC"; //CryptixCrypto";
    public static void main(String args[])
    {
        ServerSocket serSock = null;

        try
        {
            serSock = new ServerSocket(50000);
        }
        catch (IOException e)
        {
            System.err.println("Erreur de port d'écoute ! ? [" + e + "]"); System.exit(1);
        }
    }
}
```

```
Socket serSockCli = null;  
System.out.println("Serveur en attente");  
try  
{  
    serSockCli = serSock.accept();  
}
```

```

    catch (IOException e)
    {
        System.err.println("Erreur d'accept ! ? [" + e + "]"); System.exit(1);
    }

    byte b;
    DataInputStream dis=null; DataOutputStream dos=null;

    try
    {
        dis = new DataInputStream( new BufferedInputStream(  

            serSockCli.getInputStream()));  

        dos = new DataOutputStream( new BufferedOutputStream(  

            serSockCli.getOutputStream()));  

        if (dis==null || dos==null) System.exit(1);

        String user = dis.readUTF();  

        System.out.println("Utilisateur = " + user);  

        long temps = dis.readLong();  

        System.out.println("temps = " + temps);  

        double alea = dis.readDouble();  

        System.out.println("Nombre aléatoire = " + alea);  

        int longueur = dis.readInt();  

        System.out.println("Longueur = " + longueur);  

        byte[] msgD = new byte[longueur];  

        dis.readFully(msgD);

        String password = ServeurPassword.chercheMotDePasse(user);
        // confection d'un digest local
        MessageDigest md = MessageDigest.getInstance("SHA-1", codeProvider);
        md.update(user.getBytes());
        md.update(password.getBytes());
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        DataOutputStream bdos = new DataOutputStream(baos);
        bdos.writeLong(temps); bdos.writeDouble(alea);
        md.update(baos.toByteArray());
        byte[] msgDLocal = md.digest();

        // comparaison
        String réponse;
        if (MessageDigest.isEqual(msgD, msgDLocal) )
        {
            réponse = new String("OK - vous êtes connecté au serveur");
            System.out.println("Le client " + user + " est connecté au serveur");
        }
        else
        {
            réponse = new String("Sorry - votre demande de connexion est refusée");
            System.out.println("Le client " + user + " est refusé");
        }
    }
}

```

```

dos.writeUTF(réponse);
dos.flush(); // ne pas oublier ...
System.out.println("Verdict envoyé au client");
}
catch (NoSuchAlgorithmException e)
{ System.out.println("Aie aie " + e.getMessage());}
catch(NoSuchProviderException e)
{ System.out.println("Aie aie " + e.getMessage()); }
catch (EOFException e)
{ System.out.println("Aie aie EOF " + e.getMessage()); }
catch (IOException e)
{ System.err.println("Erreur ! ? [" + e + "]"); }
finally
{
try
{
dis.close();
dos.close();
serSockCli.close();
serSock.close();
System.out.println("Serveur déconnecté");
}
catch (IOException e)
{ System.err.println("Erreur ! ? [" + e + "]"); }
}
}

protected static String chercheMotDePasse (String user)
{
    return "beaugosse"; // ICI, accès à une base de données
}
}
}

```

On remarquera, pour la lecture du nombre aléatoire, l'usage de la méthode :

public void **readFully**(byte b[]) throws IOException

qui lit b.length bytes et qui donc restera bloquée tant qu'elle n'aura pu obtenir le nombre de bytes prévus. Deux exemples, l'un heureux, l'autre pas, pourraient être :

<i>client</i>	<i>serveur</i>
claude/192.168.2.2 Message à envoyer au serveur : Instanciation du message digest Envoi du message digest Réponse du serveur = OK - vous êtes connecté au serveur Client déconnecté	Serveur en attente Utilisateur = Claude temps = 985895035760 Nombre aléatoire = 0.33345827739680567 Longueur = 20 Le client Claude est connecté au serveur Verdict envoyé au client Serveur déconnecté

<i>client</i>	<i>serveur</i>
claudé/192.168.2.2 Message à envoyer au serveur : Instanciation du message digest Envoi du message digest Réponse du serveur = Sorry - votre demande de connexion est refusée Client déconnecté	Serveur en attente Utilisateur = Claude temps = 985894993630 Nombre aléatoire = 0.41053466522609194 Longueur = 20 Le client Claude est refusé Verdict envoyé au client Serveur déconnecté

Remarque

On peut reprendre le raisonnement précédent et le pousser encore un peu plus loin. En fait, on peut mémoriser dans la base de données non pas les mots de passe, mais les digests correspondants. De cette manière, une attaque réussie sur la base de données du serveur ne permettrait pas au hacker d'obtenir les mots de passe dont il aurait besoin pour usurper l'identité de clients reconnus. Les SGBD actuels permettent la mémorisation de tels digests

12. Les MACs et l'authentification légère

12.1 Le principe du HMAC

Avec la confidentialité, l'**authentification** est certainement le problème majeur de la transmission sécurisée d'informations au niveau applicatif. Rappelons que le problème est de vérifier que le message reçu provient réellement de celui que l'émetteur prétend être.

En généralisant la notion de digest, on définit un **MAC** (Message Authentication Code) comme un petit bloc de taille fixe a également pour qui a pour objectif d'assurer l'authentification et l'intégrité d'un message.

a) Technique, un MAC simple peut être un digest résultant d'un hashage (**MAC simple hashé**). Dans ce cas, comme nous l'avons vu ci-dessus, il contrôle essentiellement l'intégrité du message transmis et ne participe à un processus d'authentification que si le message considéré représente en-lui-même un élément d'authentification comme un mot de passe.

b) Mais un MAC peut aussi utiliser plutôt une clé secrète (**MAC simple à clé** - un exemple de tel MAC est CBC-MAC) : le résultat de l'application d'un chiffrement symétrique utilisant cette clé au message constitue alors le MAC. Celui-ci est envoyé au destinataire avec le message en clair. Le destinataire reçoit donc le message et le MAC. Comme il connaît la clé secrète, il peut calculer sur le message clair un second MAC. Si les deux MACs sont semblables,

- 1) cela signifie que l'intégrité est vérifiée (puisque les MACs sont semblables);
- 2) cela assure aussi de l'authentification puisqu'il faut partager le secret de la clé secrète pour vérifier cette intégrité.

c) L'analogie avec le mécanisme du digest n'a certainement pas échappé au lecteur attentif. Et, de fait, un MAC peut aussi utiliser conjointement une clé et un digest (**MAC complexe à clé et hashage**) : dans une version simple (mais il en existe de plus complexes), celui-ci est construit avec la chaîne d'entrée et cette même chaîne préalablement cryptée au moyen d'une clé secrète en plus. Le représentant typique de cette famille est le **HMAC** (keyed-Hash.Message Authentication Code), décrit dans la RFC 2104.

12.2 La programmation des HMACs

Dans les librairies cryptographiques, les macs sont des instances de la classe Mac (package javax.crypto). Le plus souvent, on crée alors un tel objet au moyen de l'habituelle factory :

```
public static final Mac getInstance(String algorithm, String provider)
    throws NoSuchAlgorithmException, NoSuchProviderException
```

et nous écrirons donc, par exemple :

```
Mac hmac = Mac.getInstance("HMAC-MD5", "CryptixCrypto");
```

La clé secrète (symétrique) à utiliser est fournie à l'objet HMAC au moyen de la méthode

```
public final void init(Key key) throws InvalidKeyException
```

Ce qui fait l'objet du "hashage crypté" est fixé par un ou plusieurs appels de la méthode :

```
public final void update(byte[] input) throws IllegalStateException
```

- donc comme pour un MessageDigest. D'ailleurs, le HMAC est finalement calculé par :

```
public final byte[] doFinal(byte[] input) throws IllegalStateException
```

L'authentification d'un message transmis par le réseau

Nous allons considérer ici qu'un programme client (par exemple – on pourrait renverser les rôles) veut communiquer dans un message à son serveur le code qu'il va utiliser durant la journée (nous pourrions le crypter, mais ce n'est pas la question ici).

L'authentification se fera au moyen du HMAC fabriqué à partir du message.

Ce HMAC est donc totalement dépendant de ce **message** et de la **connaissance de la clé symétrique** nécessaire : c'est ce dernier point qui est le fondement de l'authentification.

Les deux parties devront évidemment disposer de cette clé symétrique : nous la supposerons serialisée dans un fichier x.ser, fourni au serveur par une voie sûre (reste encore à définir celle-ci – voir plus loin).

Le client va s'écrire simplement :

ClientHMAC.java

```
package cryptographienewcryptix;

import java.io.*;
import java.net.*;
import javax.crypto.*;

public class ClientHMAC
{
    private static String codeProvider = "CryptixCrypto"; // "BC";
```

```

public static void main(String[] args)
{
    Socket cliSock = null;
    DataOutputStream dos=null; DataInputStream dis=null;

    String ligneDuServeur;
    try
    {
        cliSock = new Socket("claude", 50000);
        System.out.println(cliSock.getInetAddress().toString());
        dos = new DataOutputStream(cliSock.getOutputStream());
        dis = new DataInputStream(cliSock.getInputStream());
    }
    catch (UnknownHostException e)
    { System.err.println("Erreur ! Host non trouvé [" + e + "]"); }
    catch (IOException e)
    { System.err.println("Erreur ! Pas de connexion ? [" + e + "]"); }
    if (cliSock==null || dis==null || dos==null) System.exit(1);

    try
    {
        String Message = "Code du jour : CVCCDMMM - bye";
        System.out.println("Message a envoyer au serveur : " + Message);
        byte[] message = Message.getBytes();

        SecretKey cléSecrete;
        System.out.println("Recuperation de la cle secrète");
        ObjectInputStream cléFichS = new ObjectInputStream(
            new FileInputStream("c:\\java-netbeans-application\\x.ser"));
        cléSecrete = (SecretKey) cléFichS.readObject();
        cléFichS.close();
        System.out.println(" *** Cle secrète récupérée = " + cléSecrete.toString());

        System.out.println("Instanciation du HMAC");
        Mac hmac = Mac.getInstance("HMAC-MD5", codeProvider);
        hmac.init(cléSecrete);
        System.out.println("Hachage du message");
        hmac.update(message);
        System.out.println("Generation des bytes");
        byte[] hb = hmac.doFinal();
        System.out.println("Termine : HMAC construit");
        System.out.println("HMAC = " + new String(hb));
        System.out.println("Longueur du HMAC = " + hb.length);

        System.out.println("Envoi du message et de son HMAC");
        dos.writeUTF(Message);
        dos.writeInt(hb.length);
        dos.write(hb);

        String réponse = dis.readUTF();
    }
}

```

```
System.out.println("Reponse du serveur = " + réponse);

dos.close();dis.close(); cliSock.close();
System.out.println("Client deconnecte");

}

catch (UnknownHostException e)
{ System.err.println("Erreur ! Host non trouvé [" + e + "]"); }
catch (FileNotFoundException e)
{ System.out.println("Aie aie fichier non trouvé " + e.getMessage()); }
catch (IOException e)
{ System.err.println("Erreur ! Pas de connexion ? [" + e + "]"); }
catch (Exception e)
{ System.out.println("Aie aie imprévu ou " + e.getMessage() + " -- " + e.getClass()); }

}
```

Le serveur va appliquer la stratégie habituelle de vérification d'un digest, complétée par l'utilisation de la clé secrète connue des deux parties :

ServeurHMAC.java

```
package cryptographienewcryptix;

import java.io.*;
import java.net.*;
import java.security.*;
import javax.crypto.*;

public class ServeurHMAC
{
    private static String codeProvider = "BC"; //CryptixCrypto";
    public static void main(String args[])
    {
        ServerSocket serSock = null;
        try
        {
            serSock = new ServerSocket(50000);
        }
        catch (IOException e)
        { System.err.println("Erreur de port d'écoute ! ? [" + e + "]"); System.exit(1); }

        Socket serSockCli = null;
        System.out.println("Serveur en attente");

        try
        {
            serSockCli = serSock.accept();
        }
        catch (IOException e)
        { System.err.println("Erreur d'accept ! ? [" + e + "]"); System.exit(1); }
```

```

DataInputStream dis=null; DataOutputStream dos=null;
try
{
    dis = new DataInputStream( new BufferedInputStream(
        serSockCli.getInputStream()));
    dos = new DataOutputStream( new BufferedOutputStream(
        serSockCli.getOutputStream()));
    if (dis==null || dos==null) System.exit(1);

    String MessageRecu = dis.readUTF();
    System.out.println("Message reçu = " + MessageRecu);
    int longueur = dis.readInt();
    System.out.println("Longueur du HMAC = " + longueur);
    byte[] hmacb = new byte[longueur];
    dis.readFully(hmacb);
    byte[] message = MessageRecu.getBytes();

    System.out.println("Recuperation de la cle secrète");
    SecretKey cléSecrète;
    ObjectInputStream cléFichS = new ObjectInputStream(
        new FileInputStream("c:\\java-netbeans-application\\x.ser"));
    cléSecrète = (SecretKey) cléFichS.readObject();
    cléFichS.close();
    System.out.println("/** Clé secrète recuperée =
        "+cléSecrète.toString());

    System.out.println("Debut de verification");
    // confection d'un HMAC local
    Mac hlocal = Mac.getInstance("HMAC-MD5", codeProvider);
    hlocal.init(cléSecrète);
    System.out.println("Hachage du message");
    hlocal.update(message);
    System.out.println("Verification des HMDS");
    byte[] hlocalb = hlocal.doFinal();

    String réponse = null;
    if (MessageDigest.isEqual(hmacb, hlocalb) )
    {
        réponse = new String("OK - message authentifié");
        System.out.println("Le messsage a été authentifié");
    }
    else
    {
        réponse = new String("KO - message NON authentifié");
        System.out.println("Le messsage n'a pas été authentifié");
    }
    dos.writeUTF(réponse); dos.flush(); // ne pas oublier ...
    System.out.println("Verdict envoyé au client");
}
catch (NoSuchProviderException e)

```

```

        { System.out.println("Aie aie " + e.getMessage()); }
        catch (NoSuchAlgorithmException e)
        { System.out.println("Aie aie " + e.getMessage()); }
        catch (ClassNotFoundException e)
        { System.out.println("Aie aie " + e.getMessage()); }
        catch (InvalidKeyException e)
        { System.out.println("Aie aie " + e.getMessage()); }
        catch (EOFException e)
        { System.out.println("Aie aie EOF " + e.getMessage()); }
        catch (IOException e)
        { System.err.println("Erreur ! ? [" + e + "]"); }
    finally
    {
        try
        {
            dis.close(); dos.close(); serSockCli.close(); serSock.close();
            System.out.println("Serveur deconnecte");
        }
        catch (IOException e)
        { System.err.println("Erreur ! ? [" + e + "]"); }
    }
}
}

```

Un exemple de test donne sur les consoles respectives :

<i>client</i>	<i>serveur</i>
claudé/192.168.2.2 Message à envoyer au serveur : Code du jour : CVCCDMMM - bye Recuperation de la cle secrète *** Cle secrète récupérée = cryptix.jce.provider.key.RawSecretKey@14e d9ff Instanciation du HMAC Hachage du message Generation des bytes Termine : HMAC construit HMAC = $\text{TM}^{\text{I}}\text{O}^{\text{O}}\text{y}^{\text{d}}\text{?}^{\text{a}}\text{?}^{\text{?}}\text{?}^{\text{z}}\text{?}^{\text{t}}\text{?}^{\text{3}}\text{?}^{\text{4}}\text{?}^{\text{2}}$ Longueur du HMAC = 16 Envoi du message et de son HMAC Reponse du serveur = OK - message authentifié Client deconnecte	Serveur en attente Message reçu = Code du jour : CVCCDMMM - bye Longueur du HMAC = 16 Recuperation de la cle secrète *** Clé secrète recuperée = cryptix.jce.provider.key.RawSecretKey@13f3789 Debut de verification Hachage du message Verification des HMACS Le messsage a été authentifié Verdict envoyé au client Serveur deconnecte

Le mécanisme d'authentification considéré ici qualifié d'"**authentification légère**" : elle ne fait intervenir que des algorithmes peu coûteux en opérations machines, principalement des chiffrements symétriques. Bien sûr, les reproches que l'on peut formuler à

la cryptographie symétrique en opposition à la cryptographie asymétrique (comme la fragilité du secret de la clé symétrique partagée) se reportent sur l'authentification par HMAC ...

Et il existe aussi un mécanisme d'authentification plus robuste, plus "lourd", basé sur cette cryptographie asymétrique : le mécanisme de la signature électronique. Voyons cela ...

Remarque

Plutôt que de déserialiser la clé secrète, on peut imaginer de la générer à partir d'une phrase secrète (à la rigueur un simple mot de passe) partagé par les deux parties. Ceci peut se faire en utilisant la classe SecretKeySpec (package javax.crypto.spec) qui est capable de construire une clé symétrique à partir d'un tableau de byte, indépendamment de tout provider.

```
public SecretKeySpec(byte[] key, String algorithm)
```

Dans les programmes précédents, on remplacera alors la déserialisation de la clé par une séquence du type suivant :

```
// Fabrication de la clé secrète à partir de la phrase secrète
SecretKey cléSecrete;
System.out.println("Entrez la phrase secrète : ");
BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
String phraseSecrète = in.readLine();
cléSecrete = new SecretKeySpec(phraseSecrète.getBytes(), "DES");
System.out.println(" *** Cle secrète générée = " + cléSecrete.toString());
```

Ce qui donnera à l'exécution l'entrée au clavier de la phrase mystère :

<i>client</i>	<i>serveur</i>
claudie/192.168.2.2 Message à envoyer au serveur : Code du jour : CVCCDMMM - bye Entrez la phrase secrète : <i>John 5eme comte du Devonshire</i> *** Cle secrète générée = javax.crypto.spec.SecretKeySpec@11da5 Instanciation du HMAC Hachage du message Génération des bytes Termine : HMAC construit HMAC = i^?ó e^34#jn^ v4Tx^ Longueur du HMAC = 16 Envoi du message et de son HMAC Reponse du serveur = OK - message authentifié Client déconnecté	Serveur en attente Message reçu = Code du jour : CVCCDMMM - bye Longueur du HMAC = 16 Entrez la phrase secrète : <i>John 5eme comte du Devonshire</i> *** Cle secrète générée = javax.crypto.spec.SecretKeySpec@11da5 Début de vérification Hachage du message Vérification des HMACS Le message a été authentifié Verdict envoyé au client Serveur déconnecté

13. Les signatures électroniques et l'authentification lourde

13.1 Le principe de la signature électronique

Dans la vie courante, un acte authentique de notaire est couvert de signatures et de paraphes. Dans la foulée, on a donc imaginé d'orner les informations sensibles d'une "signature électronique" :

une **signature électronique** est un bloc de données qui a été créé en utilisant une clé privée et qui peut être vérifié par la clé publique correspondante sans la connaissance de la clé privée.

Le terme "signature électronique" possède des synonymes : on parle aussi de "signature numérique" ou encore de "**signature digitale**", ce dernier qualificatif étant particulièrement utilisé dans les pays anglo-saxons [*digital*].

13.2 La construction d'une signature basée sur un digest

Pour construire une signature,

- 1) on construit un **message digest** du document à envoyer, selon un algorithme de digest (par exemple, SHA-1);
- 2) on chiffre, au moyen d'un algorithme approprié de chiffrement asymétrique (par exemple, RSA), la chaîne obtenue pour le digest en utilisant la **clé privée** du signataire. Ce que l'on obtient est la **signature électronique ou digitale**.

L'expéditeur envoie alors le message et la signature. Le destinataire :

- 1) déchiffre la signature au moyen de la clé publique du signataire du message; il obtient ainsi le message digest calculé lors de l'envoi;
- 2) calcule un second message digest pour le message obtenu; si les deux digestes correspondent, on peut être assuré que l'information n'a pas été falsifiée, ce qui assure donc l'**intégrité**; dans ce cas aussi, la clé privée utilisée était bien celle du signataire, ce qui assure donc aussi l'**authentification**;
- 3) peut convaincre un tiers que le document reçu a bien été envoyé par le propriétaire de la clé privée, pas par quelqu'un d'autre : on résout ainsi le problème de la **non-répudiation**.

Par rapport à la notion de signature manuscrite, une nuance est d'importance et mérite d'être bien soulignée :

Une signature électronique est fonction, outre de la **clé privée** utilisée, du **message** qu'elle accompagne : **elle est donc propre non seulement au propriétaire de la clé privée mais aussi à ce message**.

En pratique, ceci signifie donc qu'un hacker qui aurait dérobé une signature électronique ne saurait s'en servir pour signer un autre message ...

Remarque

La technique ainsi exposée peut aussi servir pour envoyer une clé publique en assurant qu'elle appartient bien à celui que l'on croit.

13.3 Un exemple

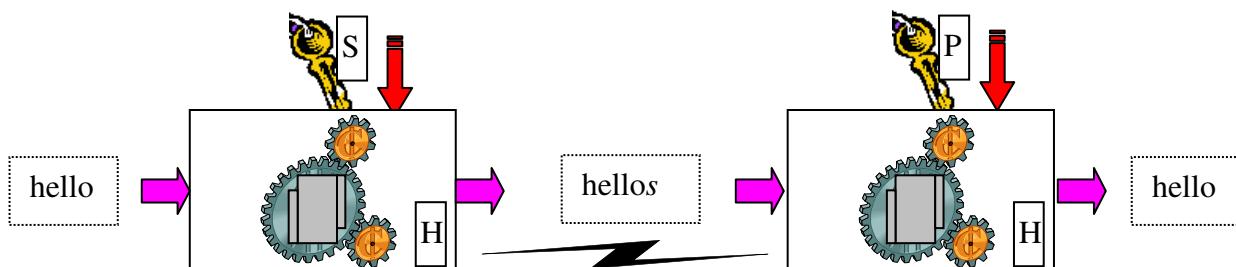
Supposons encore une fois que le message à transmettre par James à Francesca soit simplement "hello". L'expéditeur (James) et le destinataire (Francesca) sont toujours d'accord sur la méthode de hachage :

$$h(\text{message}) = (\sum (\text{codes ASCII des caractères})) \% 67$$

De plus, James va signer son message pour que Francesca soit bien certaine que c'est l'élu de son cœur qui la salut.

message digest d'origine :

message	"hello"	0110 1000	0110 0101	0110 1100	0110 1100	0110 1111
valeur de hachage = $h1 = (104+101+108+108+111) \% 67 = 63$						
clé privée de James = $(n,d) = (3233, 2753)$						
signature = $s = (h1^d) \% n = (63^{2753}) \% 3233 = 1393$						
concaténation						
message avec digest	"hello"s	104	101	108	108	111
						1393



message digest à l'arrivée :

message avec digest	"hello"s	104	101	108	108	1111	1393
clé publique de James = $(3233, 17)$							
$s = 1393$							
$h1 = (1393^{17}) \% 3233 = 63$							
$h2 = \text{valeur de hachage} = (104+101+108+108+111) \% 67 = 63$							
conclusion							
$h1 = h2 \rightarrow$ le message n'a pas été modifié + c'est bien James qui l'a envoyé							

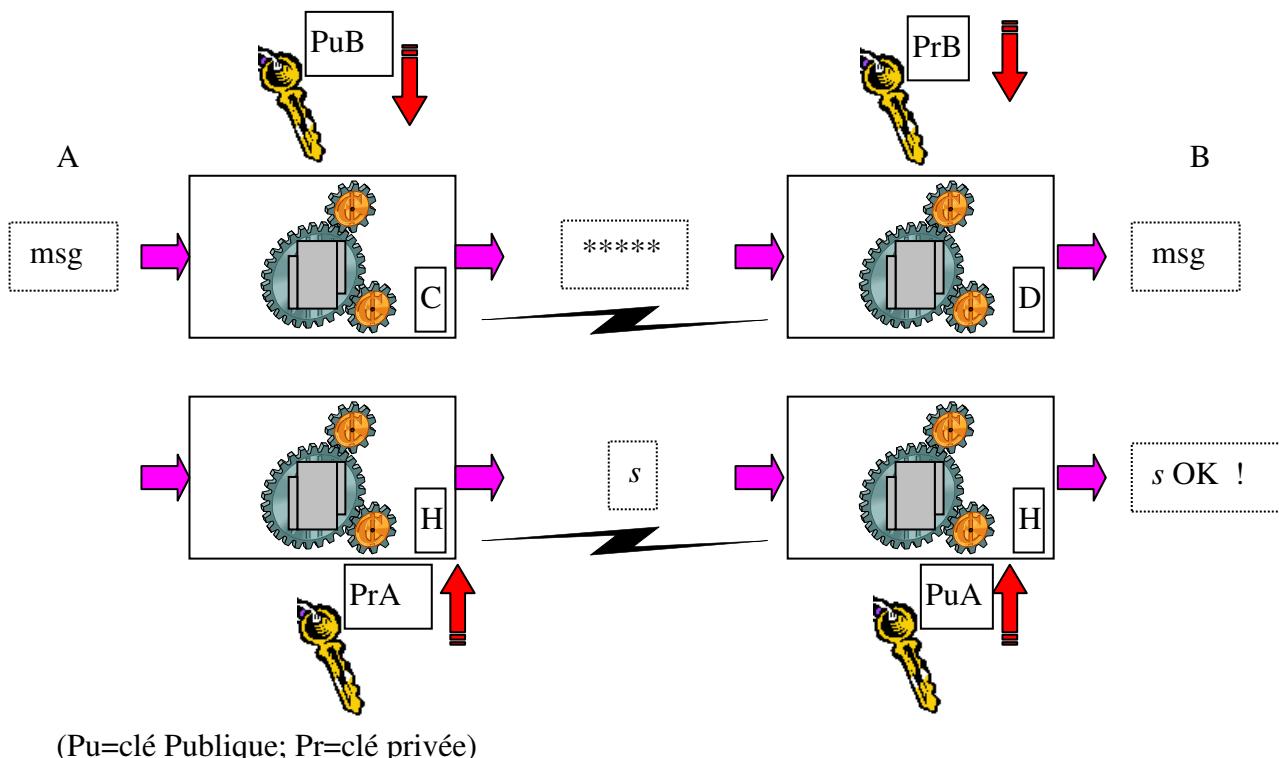
Remarque

Evidemment, tout ceci suppose que le message (ici, "hello") est compréhensible par le destinataire. Si le contenu n'a rien de confidentiel, le problème ne se pose pas. Par contre, si le message doit rester secret, il doit être crypté pour être envoyé, indépendamment de la confection et de l'envoi de la signature.

Supposons donc que B veuille recevoir un message codé de A et vérifier la signature de ce même A. La succession des opérations est :

- ◆ A envoie à B un message crypté au moyen de la clé publique de B (à condition d'être sûr que c'est bien celle de B → voir plus loin les certificats);
- ◆ B décrypte le message au moyen de sa clé privée;
- ◆ A calcule le digest du message et le crypte au moyen de sa clé privée : il construit ainsi sa signature qu'il envoie à B;

- ◆ B décrypte la signature au moyen de la clé publique de A; il obtient ainsi le digest envoyé en principe par A;
- ◆ B calcule, avec la même fonction de hachage, le digest du message reçu;
- ◆ si les deux digestes sont égaux, la signature est bien vérifiée.



13.4 Quelques algorithmes de signatures digitales courants

Essentiellement, on connaît :

- ◆ **DSA (Digital Signature Algorithm)** : il utilise une clé de 1024 bits; c'est le standard du NIST, sous le nom de DSS (Digital Signature Standard); cet algorithme utilise les 3 paramètres publics suivants :

- p: nombre premier dont le nombre de bits est un multiple de 64 et est compris entre 512 et 1024;
- q: facteur premier à 160 bits de p-1;
- g: calculé par $h^{(p-1)/q} \% p$, où h est un nombre quelconque inférieur à p-1 tel que $h^{(p-1)/q} \% p > 1$.

La clé privée est alors donnée par $x =$ un nombre plus petit que q tandis que la clé publique est donnée par $y = g^x \% p$.

L'expéditeur crée sa signature digitale sous la forme de deux nombres r et s au moyen d'un nombre aléatoire k et des formules

$$r = (g^k \% p) \% q$$

$$s = (k^{-1}(\text{digest} + x \cdot r)) \% q$$

On nous fera grâce des formules de vérification (ouf ;-)) ...

A remarquer que si on n'applique pas de digest, on obtient *un algorithme de cryptage asymétrique* dont le texte crypté résultant est r ☺.

- ♦ **RSA** : il s'agit de l'algorithme de chiffrement asymétrique décrit plus haut appliqué aux signatures. L'expéditeur crée sa signature digitale au moyen de la formule

$$\text{signature} = (<\text{digest}>^d) \% n$$

où (n,d) est sa clé privée. Il envoie ensuite le message et la signature à un destinataire. Celui-ci utilise la clé publique correspondante (n,e) pour calculer :

$$\text{digest} = (<\text{signature}>^e) \% n$$

ce qui lui permet de comparer les résultats.

13.5 La signature d'un message transmis par le réseau

Nous allons à nouveau considérer qu'un programme client veut communiquer dans un message à son serveur le code qu'il va utiliser durant la journée (toujours sans le crypter). L'authentification se fera au moyen de la signature envoyée, **signature fabriquée à partir du message** et donc totalement attachée à ce message (rien n'empêche de reprendre en plus des composantes évoquées dans le paragraphe consacré aux digests comme un nombre aléatoire ou une date).

Cette signature est donc totalement dépendante de ce **message** et de la **connaissance de la clé privée** nécessaire : c'est le fondement de ce type d'authentification.

En ce qui concerne le **client**, il va donc falloir tout d'abord, d'après ce qui précède, se procurer une paire de clés (clé publique, clé privée). Un petit programme PaireDeCles (dont la première partie est semblable à celle vue au paragraphe 6) nous permettra de générer une telle paire selon l'algorithme RSA pour ensuite les sérialiser dans des fichiers xp.ser et xs.ser (placés dans un répertoire nommé "clés"). Le client pourra ainsi constamment se procurer sa clé privée et le serveur devra donc acquérir la clé publique d'une manière ou d'une autre. Ici, nous supposerons que le fichier de sérialisation xp.ser a été fourni au serveur par une voie sûre (nous reparlerons de ce problème avec les certificats).

PaireDeCles.java

```
import java.io.*;
import java.security.*;

public class PaireDeCles
{
    private static String codeProvider = "BC"; //CryptixCrypto";
    private static final SecureRandom prng = new SecureRandom();

    public static void main(String args[])
    {
        try
        {

```

```

// Génération des clés
System.out.println("Tentative d'obtention d'un generateur de cle");
KeyPairGenerator genCles = KeyPairGenerator.getInstance("RSA",
    codeProvider);
System.out.println("Tentative d'initialisation du generateur de cle");
int se = 512; // par exemple
genCles.initialize(se, prng);
System.out.println("Tentative d'obtention de cles");
KeyPair deuxCles = genCles.generateKeyPair();
PublicKey cléPublique = deuxCles.getPublic();
PrivateKey cléPrivee = deuxCles.getPrivate();
System.out.println(" *** Cle publique generee = " + cléPublique);
System.out.println(" *** Cle privee generee = " + cléPrivee);

// Sérialisation de clés
System.out.println(" *** Cle publique generee serialisee");
ObjectOutputStream cléPubliqueFich = new ObjectOutputStream(
    new FileOutputStream("c:\\java-sun-application\\cles\\xp.ser"));
System.out.println("fichier ouvert");
cléPubliqueFich.writeObject(cléPublique);
System.out.println("cle ecrite");
cléPubliqueFich.close();
System.out.println(" *** Cle privee generee serialisee");
ObjectOutputStream cléPrivéeFich = new ObjectOutputStream(
    new FileOutputStream("c:\\java-sun-application\\cles\\xs.ser"));
cléPrivéeFich.writeObject(cléPrivee);
cléPrivéeFich.close();

}

catch (NoSuchAlgorithmException e)
{ System.out.println("Aie aie " + e.getMessage()); }
catch (FileNotFoundException e)
{ System.out.println("Aie aie fichier non trouvé " + e.getMessage()); }
catch (Exception e)
{ System.out.println("Aie aie imprévu ou " + e.getMessage() + " -- " + e.getClass()); }

}

```

Ensuite, nous pouvons écrire le client qui va construire sa signature digitale sur son message, instance la classe abstraite **Signature** (java.security). Pour cela, il lui faudra :

1) obtenir une instance de signature (qui utilise un digest SHA-1 et un algorithme RSA pour la clé) au moyen de l'une des méthodes factory :

```
public static Signature getInstance(String algorithm) throws NoSuchAlgorithmException  
public static Signature getInstance(String algorithm, String provider)  
    throws NoSuchAlgorithmException
```

Dans notre cas, par exemple :

```
Signature s = Signature.getInstance("SHA1withRSA", "CryptixCrypto");
```

2) initialiser l'objet signature en lui fournissant la clé privée à utiliser; ceci se fait au moyen de la méthode

```
public final void initSign(PrivateKey privateKey) throws InvalidKeyException
```

Pour ce qui nous concerne, si la clé privée a été obtenue par :

```
ObjectInputStream cléFichS = new ObjectInputStream(  
    new FileInputStream("c:\\java-sun-application\\cles\\xs.ser"));  
PrivateKey cléPrivee = (PrivateKey) cléFichS.readObject();  
cléFichS.close();
```

L'initialisation de la signature se fera alors par :

```
s.initSign(cléPrivee);
```

3) préparer la signature parler en y plaçant l'"ingrédient" (ici, le message) au moyen de la méthode

```
public final void update(byte data[]) throws SignatureException
```

qui permet d'ajouter à la signature le tableau de bytes passé comme paramètre. Il suffira évidemment ici d'utiliser la méthode getBytes.

4) enfin, appeler la méthode qui génère effectivement la signature :

```
public final byte[] sign() throws SignatureException  
et envoyer au serveur le message et la signature. Le client peut donc s'écrire :
```

ClientSignature.java

```
import java.io.*;  
import java.net.*;  
import java.security.*;  
import javax.crypto.*;  
  
public class ClientSignature  
{  
    private static String codeProvider = "CryptixCrypto"; // "BC";  
    public static void main(String[] args)  
    {  
        Socket cliSock = null; DataOutputStream dos=null; DataInputStream dis=null;  
        String ligneDuServeur;  
        try  
        {  
            cliSock = new Socket("claude", 50000);  
            System.out.println(cliSock.getInetAddress().toString());  
            dos = new DataOutputStream(cliSock.getOutputStream());  
            dis = new DataInputStream(cliSock.getInputStream());  
        }  
    }
```

```

        catch (UnknownHostException e)
        { System.err.println("Erreur ! Host non trouvé [" + e + "]"); }
        catch (IOException e)
        { System.err.println("Erreur ! Pas de connexion ? [" + e + "]"); }
        if (cliSock==null || dis==null || dos==null) System.exit(1);
        try
        {
            String Message = "Code du jour : CVCCDMMM - bye";
            System.out.println("Message a envoyer au serveur : " + Message);
            byte[] message = Message.getBytes();

            PrivateKey cléPrivée;
            System.out.println("Recuperation de la cle privee");
            ObjectInputStream cléFichS = new ObjectInputStream(
                new FileInputStream("c:\\java-sun-application\\cles\\xs.ser"));
            cléPrivée = (PrivateKey) cléFichS.readObject();
            cléFichS.close();
            System.out.println(" *** Cle privee recuperée = " + cléPrivée.toString());

            System.out.println("Instanciation de la signature");
            Signature s = Signature.getInstance("SHA1withRSA",codeProvider);
            System.out.println("Initialisation de la signature");
            s.initSign(cléPrivée);
            System.out.println("Hachage du message");
            s.update(message);
            System.out.println("Generation des bytes");
            byte[] signature = s.sign();
            System.out.println("Termine : signature construite");
            System.out.println("Signature = " + new String(signature));
            System.out.println("Longueur de la signature = " + signature.length);
            System.out.println("Envoi du message et de la signature");
            dos.writeUTF(Message);
            dos.writeInt(signature.length);
            dos.write(signature);

            String réponse = dis.readUTF();
            System.out.println("Reponse du serveur = " + réponse);
            dos.close();dis.close(); cliSock.close();
            System.out.println("Client deconnecte");
        }
        catch (UnknownHostException e)
        { System.err.println("Erreur ! Host non trouvé [" + e + "]"); }
        catch (FileNotFoundException e)
        { System.out.println("Aie aie fichier non trouvé " + e.getMessage()); }
        catch (IOException e)
        { System.err.println("Erreur ! Pas de connexion ? [" + e + "]"); }
        catch (Exception e)
        { System.out.println("Aie aie imprévu ou " + e.getMessage() + " -- " + e.getClass()); }
    }
}

```

En ce qui concerne le **serveur**, il devra se procurer la clé publique associée à la clé privée utilisée par le client. Evidemment, ceci est incontournable, mais cette clé publique doit être obtenue par un moyen sûr (comme annoncé, nous y reviendrons avec le paragraphe consacré aux certificats). Pour notre exemple, nous supposerons que le client et le serveur se connaissent suffisamment pour que le serveur ait reçu la clé publique sérialisée dans un fichier xp.ser.

Donc, notre serveur :

1) se procure la clé publique du client :

```
ObjectInputStream cléFichP = new ObjectInputStream(  
    new FileInputStream("c:\\\\ java-sun-application\\\\cles\\\\xp.ser"));  
PublicKey cléPublique = (PublicKey) cléFichP.readObject();  
cléFichP.close();
```

2) génère lui aussi une signature s de type SHA-1/RSA et l'initialise également, mais pour la vérification, au moyen de la méthode :

```
public final void initVerify(PublicKey publicKey) throws InvalidKeyException
```

ce qui donne ici :

```
s.initVerify(cléPublique);
```

3) prépare la signature parler en y plaçant l'"ingrédient" qu'est le tableau de bytes correspondant au message envoyé par l'utilisateur qui s'est connecté au moyen de la méthode **update()**;

4) appelle la méthode de test de la signature :

```
public final boolean verify(byte signature[]) throws SignatureException  
donc ici :
```

```
boolean ok = s.verify(signature);
```

si signature est la signature reçue avec le message. Notre serveur s'écrit donc :

ServeurSignature.java

```
import java.io.*;  
import java.net.*;  
import java.security.*;  
public class ServeurSignature  
{  
    private static String codeProvider = "BC"; //CryptixCrypto";  
    public static void main(String args[])  
    {  
        ServerSocket serSock = null;
```

```

try
{
    serSock = new ServerSocket(50000);
}
catch (IOException e)
{
    System.err.println("Erreur de port d'écoute ! ? [" + e + "]");
    System.exit(1);
}

Socket serSockCli = null;
System.out.println("Serveur en attente");

try
{
    serSockCli = serSock.accept();
}
catch (IOException e)
{
    System.err.println("Erreur d'accept ! ? [" + e + "]");
    System.exit(1);
}
DataInputStream dis=null; DataOutputStream dos=null;

try
{
    dis = new DataInputStream( new BufferedInputStream(
        serSockCli.getInputStream()));
    dos = new DataOutputStream(new BufferedOutputStream(
        serSockCli.getOutputStream()));
    if (dis==null || dos==null) System.exit(1);

    String MessageRecu = dis.readUTF();
    System.out.println("Message reçu = " + MessageRecu);
    int longueur = dis.readInt();
    System.out.println("Longueur de la signature = " + longueur);
    byte[] signature = new byte[longueur];
    dis.readFully(signature);
    byte[] message = MessageRecu.getBytes();

    System.out.println("Recuperation de la cle publique");
    PublicKey cléPublique;
    ObjectInputStream cléFichP =
        new ObjectInputStream(new
            FileInputStream("c:\\java-sun-application\\cles\\xp.ser"));
    cléPublique = (PublicKey) cléFichP.readObject();
    cléFichP.close();
    System.out.println("/** Cle publique recuperée = "+cléPublique.toString());

    System.out.println("Debut de verification de la signature construite");
}

```

```

// confection d'une signature locale
Signature s = Signature.getInstance("SHA1withRSA", codeProvider);
s.initVerify(cléPublique);
System.out.println("Hachage du message");
s.update(message);
System.out.println("Verification de la signature construite");
boolean ok = s.verify(signature);
String réponse;

if (ok) réponse = new String("Signature testee avec succes");
else réponse = new String("Signature testee sans succes");
System.out.println(réponse);
dos.writeUTF(réponse); dos.flush(); // ne pas oublier ...
System.out.println("Verdict envoyé au client");

}

catch (NoSuchProviderException e)
{ System.out.println("Aie aie " + e.getMessage()); }
catch (NoSuchAlgorithmException e)
{ System.out.println("Aie aie " + e.getMessage()); }
catch (ClassNotFoundException e)
{ System.out.println("Aie aie " + e.getMessage()); }
catch (InvalidKeyException e)
{ System.out.println("Aie aie " + e.getMessage()); }
catch (SignatureException e)
{ System.out.println("Aie aie " + e.getMessage()); }
catch (EOFException e)
{ System.out.println("Aie aie EOF " + e.getMessage()); }
catch (IOException e)
{ System.err.println("Erreur ! ? [" + e + "]"); }
finally
{
    try
    {
        dis.close(); dos.close();
        serSockCli.close(); serSock.close();
        System.out.println("Serveur deconnecte");
    }
    catch (IOException e)
    { System.err.println("Erreur ! ? [" + e + "]"); }
}
}
}

```

Un exemple de test donne sur les consoles respectives :

<i>client</i>	<i>serveur</i>
claudé/192.168.2.2 Message à envoyer au serveur : Code du jour : CVCCDMMM - bye Recuperation de la clé privée *** Cle privée récupérée = RSA Private CRT Key modulus: 9159b68307...d82be46434fe9dd public exponent: 10001 private exponent: 4a6b...72e8cd356ce2895 ... Instanciation de la signature Initialisation de la signature Hachage du message Génération des bytes Termine : signature construite Signature = + ?íSbps'ûš<‡gÈ½ øŠ? @j#•—cG}ã IFD¼Ó£Zðip> ð"é2IXIå8Ó âÔ@'™k h...j4 Longueur de la signature = 64 Envoi du message et de la signature Réponse du serveur = Signature testée avec succès Client déconnecté	Serveur en attente Message reçu = Code du jour : CVCCDMMM - bye Longueur de la signature = 64 Recuperation de la clé publique *** Cle publique récupérée = <-----RSA PublicKey: modulus: 9159b68307...d82be46434fe9dd public exponent: 10001 ... Début de vérification de la signature construite Hachage du message Vérification de la signature construite Signature testée avec succès Verdict envoyé au client Serveur déconnecté

14. Les certificats

14.1 Le principe

Dans un algorithme asymétrique, la connaissance de la clé publique est incontournable et fondamentale pour la communication. Bien sûr, on pourrait acquérir la clé publique d'un destinataire sur un serveur, mais

- ◆ comment savoir si on obtient la vraie clé ?
- ◆ comment savoir qu'il s'agit bien de la clé de la personne visée ?

On pourrait penser à de grandes bases de données publiques contenant les paires "clé publique - propriétaire" – mais se poserait alors le problème de la sécurité de telles bases. Que se passerait-il, en effet, si un hacker parvenait à placer dans la base de données, pour une victime choisie, une clé publique fabriquée par ses soins ? On a donc imaginé une autre technique.

En utilisant une signature digitale, on peut certifier qu'une clé publique appartient bien à une certaine personne en créant un ainsi ce que l'on appelle un **certificat**, c'est-à-dire un message qui a valeur officielle et qui a la reconnaissance de tous (exactement comme un permis de conduire ou un acte authentique notarié). Plus précisément, on construit un message comportant :

- ◆ l'identification de son propriétaire;
- ◆ la clé publique;

et ce message est ensuite signé au moyen de la clé d'un organisme en qui on a toute confiance : cet organisme est, dirait-on juridiquement, le "*tiers de confiance*", analogue à un notaire, un juge, une banque, une administration. Ces "racines de la hiérarchie de confiance", que l'on appelle les *Certification Authority (CA)*, sont répandues à travers dans le monde; une société bien connue dans ce domaine est Verisign (<http://www.verisign.com>). Le message ainsi complété de l'identification du CA et de sa signature constitue ce que l'on appelle un **certificat d'authentification**, dont le rôle est donc de servir d'attestation officielle que la clé publique en question est bien celle d'une personne précise :

certificat : informations fondamentales
<ul style="list-style-type: none">◆ identification du propriétaire◆ clé publique◆ identification du CA◆ signature digitale du CA

A priori, le contenu d'un certificat peut être variable dans une certaine mesure. La nécessité d'une norme est vite apparue; la norme la plus usitée est dénommée **X.509** et stipule qu'un certificat comporte :

certificat X509 : informations plus complètes
<ul style="list-style-type: none">◆ numéro de version◆ numéro de série◆ identifiant de l'algorithme de signature◆ identification du propriétaire de la clé publique certifiée◆ période de validité◆ identification du CA (son DN)◆ information sur l'algorithme de clé publique◆ clé publique◆ extensions diverses◆ signature digitale du CA pour les champs ci-dessus

On utilise actuellement la version 3 de cette norme, qui ne diffère de ses prédécesseurs que par les extensions.

Reste encore une question. Que se passe-t-il si le propriétaire de ce la clé publique certifiée perd sa clé privée (ou se la fait voler) ? Il est bien entendu impérieux de révoquer le certificat (dans le cas du vol, quelqu'un peut se faire passer pour le propriétaire sans difficultés). Pour ce faire, les CA gèrent des *Certificate Revocation List (CRL)*, analogues aux listes des cartes de crédit non valides. Il est également possible de s'adresser à un serveur **OCSP (Online Certification Status Protocol)** chargé de répondre aux demandes de vérification de validité.

14.2 Vérifier un certificat

Comment savoir si le certificat que l'on vient de recevoir, et qui fournit la clé publique de quelqu'un, est bien le vrai ? Bien sûr, en vérifiant la signature du CA. Comme nous l'avons vu dans le paragraphe consacré aux signatures, on va d'une part calculer le digest des informations contenues dans le certificat (au moyen de l'algorithme de hachage correspondant au type de signature – il est spécifié dans le certificat) et d'autre part utiliser la clé publique du

CA pour obtenir le digest qui est à la base de la signature : si les deux digests coïncident, la signature est donc déclarée valide.

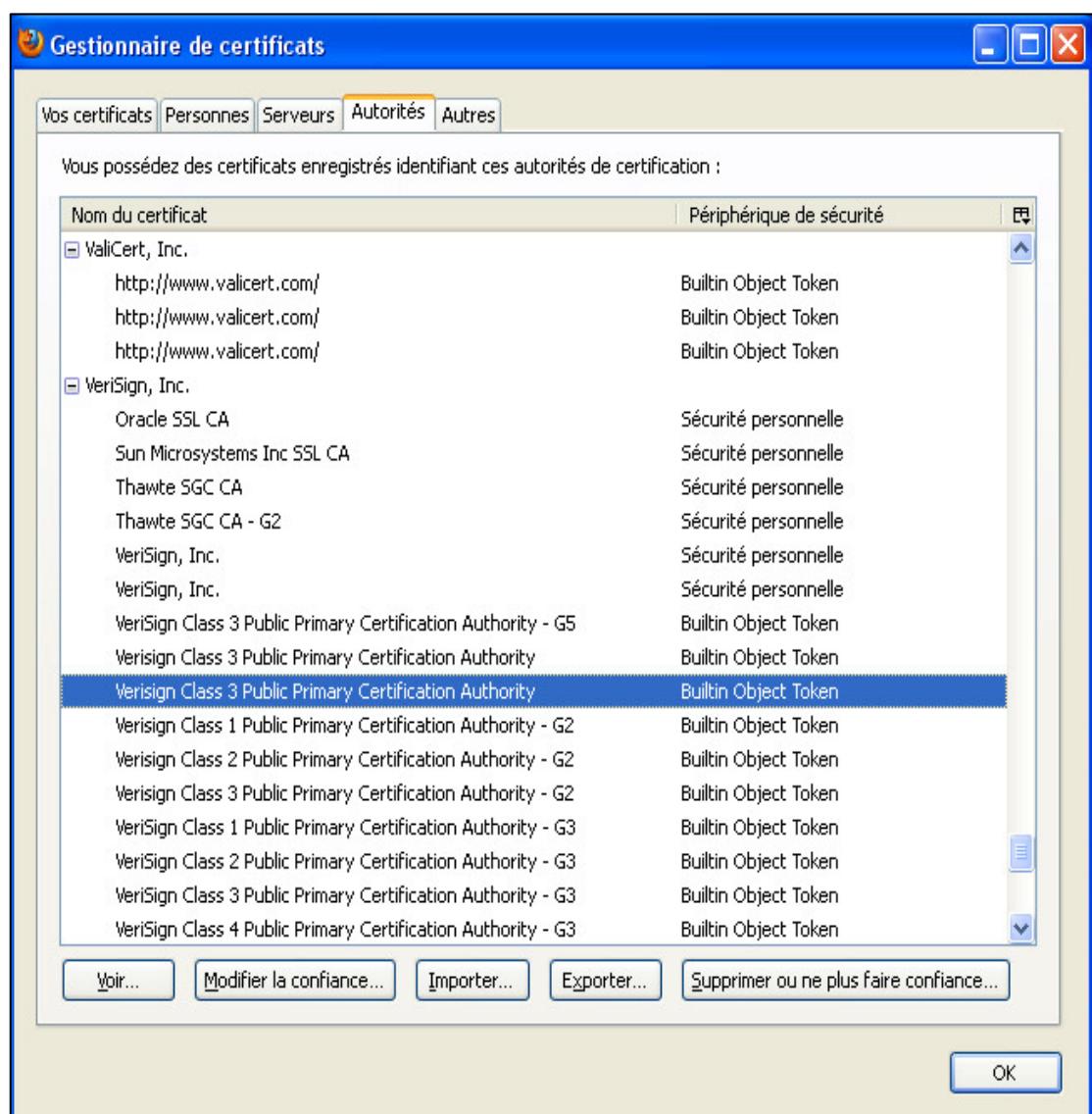
Pour vérifier cette signature, **il faut donc disposer de la clé publique du CA**. Deux possibilités :

- ◆ le CA est bien connu (c'est une autorité bien connue de niveau mondial) : sa clé publique est alors bien connue aussi et la vérification peut se faire sans problèmes;
- ◆ le CA est seulement local; il faut alors se procurer un certificat pour ce CA local, certificat signé par un autre CA plus connu; il faut donc alors recommencer les opérations pour ce nouveau certificat; et ainsi de suite jusqu'à parvenir à un CA bien connu – on parle encore de "chaîne de certificats".

Les browsers possèdent toute une série de clés publiques, sous forme de certificats. Ceux qui s'y trouvent par défaut sont évidemment ceux des CA comme AOL, Globalsign ou surtout Verisign. On peut le vérifier par exemple dans Firefox par :

Outils →
Options →
Avancé →
Chiffrement
→ Afficher
les
certificats

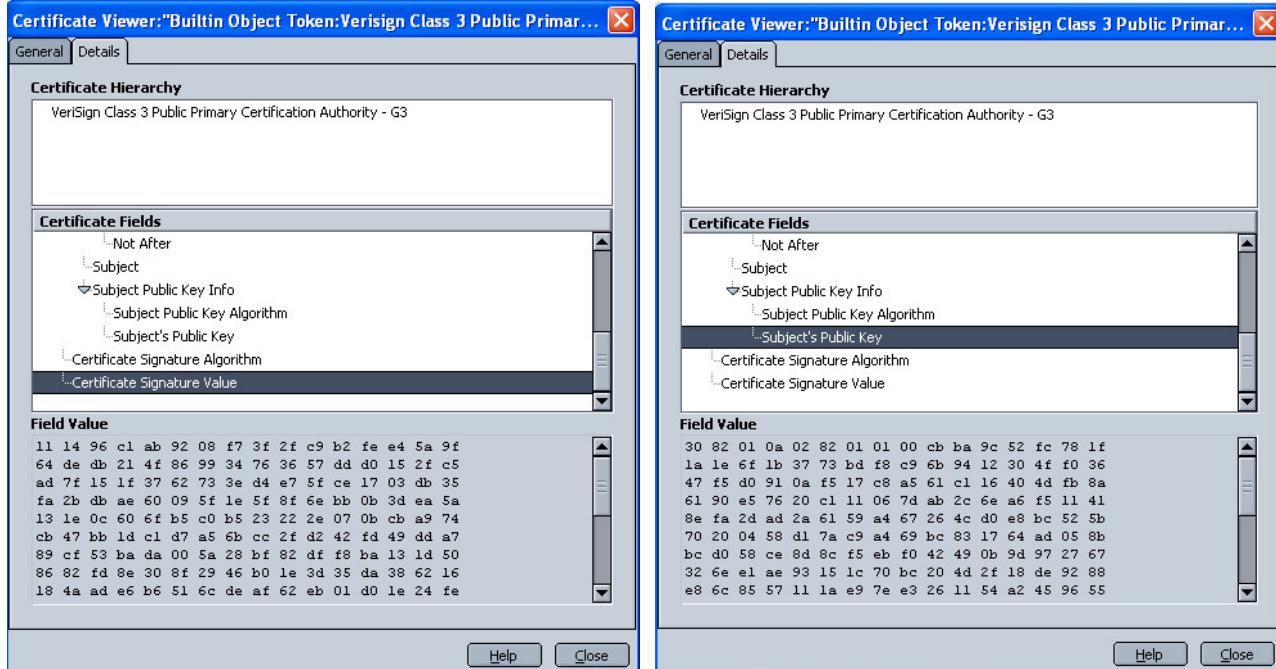
qui donne :



La sélection d'un certificat quelconque (par exemple, "Verisign Class 3 Public Primary Certification Authority – G3") donne plus de renseignements, du genre :



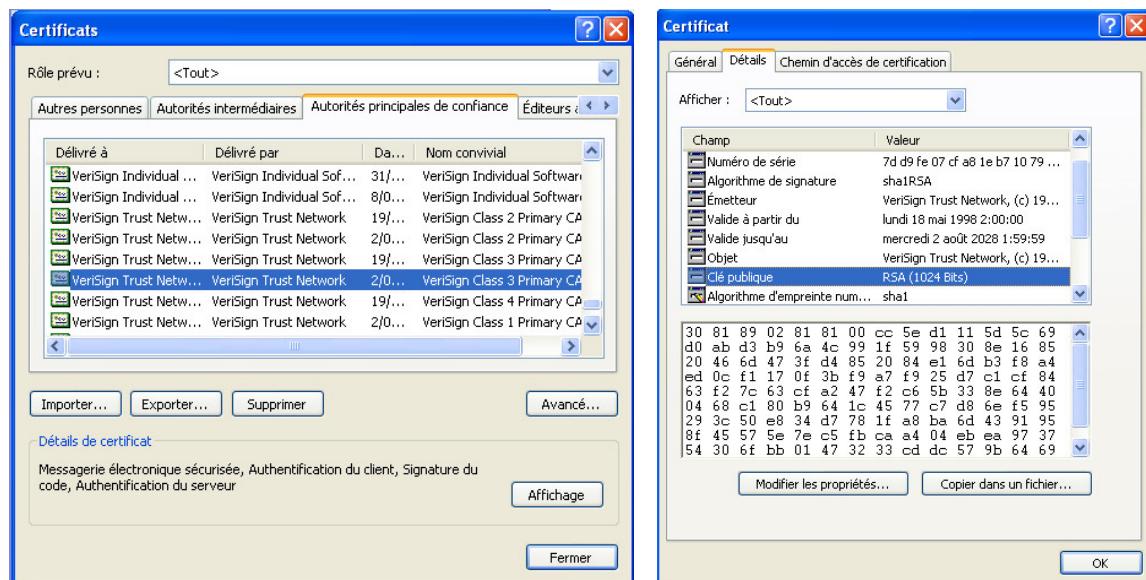
L'onglet "Details" fournissant notamment la nature des algorithmes utilisés, la clé publique faisant l'objet du certificat et la signature du CA:



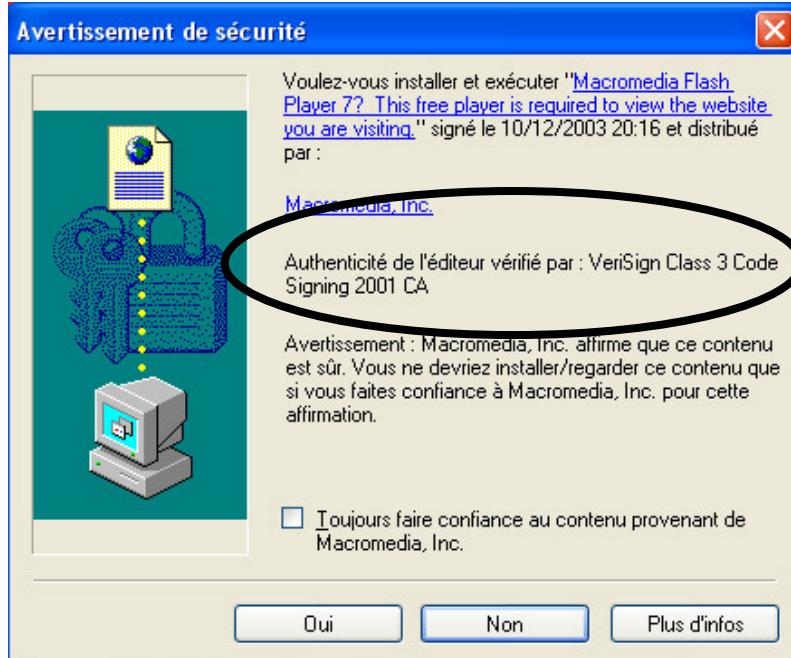
Dans Internet Explorer, on suit :

Outils → Options Internet → Contenu → Certificats

et on trouve évidemment des renseignements similaires :



Avec de telles informations, on peut sans problème vérifier qu'un serveur quelconque est bien celui qu'il prétend. Il lui suffit en effet d'envoyer son certificat et le browser pourra vérifier si il est signé par un CA digne de confiance :



Remarque

Un certificat peut être enregistré selon divers formats. Les plus courants sont **DER** (Definite Encoding Rule) avec un encodage en notation ASN.1 (.der, .cer, .crt, .cert) et **PEM** (Privacy Enhanced Mail) qui est un format DER encodé en base64 auquel sont ajoutés des en-têtes en ASCII (voir plus loin).

14.3 La gestion des clés

La gestion des certificats est malingre en JDK 1.1 : elle se limite à l'interface **Certificate** qui se trouve dans le package `java.security`. Le JDK 1.2 a apporté de sérieuses améliorations en apportant une nouvelle classe appelée également **Certificate**, mais dans le package `java.security.cert`. Bien sûr, elle comporte la méthode

```
public abstract PublicKey getPublicKey()
```

qui fournit la clé publique contenue dans le certificat. Nous en reparlerons plus loin. Pour l'instant, intéressons-nous à la classe **KeyStore**, apparue avec le **JDK 1.2** dans le package `java.security`. Elle permet de réaliser la gestion des clés. En fait, il s'agit d'une sorte de dictionnaire qui contient, de manière cryptée, **deux types d'entrées** :

- ◆ d'une part (**Key Entry**), une clé privée et une liste de certificats concernant la clé publique correspondante; on peut ainsi prouver qui l'on est; rien n'interdit donc de disposer de plusieurs couples clé publique/clé privée, destinées à des usages différents (par exemple, un couple pour signer ses applications, un couple pour le e-commerce, etc);
- ◆ d'autre part (**Trusted Certificate Entry**), un certificat d'une personne considérée comme sûre; on peut ainsi authentifier une autre partie.

Ces informations sont désignées, au sein de la structure, par des identifiants appelés des *alias* qui permettront de retrouver l'information demandée (comme une clé – mot malheureux ici – dans une table de hachage). Ainsi, de manière schématique, un objet KeyStore exemple ressemble à ceci :

<i>alias</i>	<i>information</i>	<i>usage</i>
cleApplets	clé privée + clé publique + certificat(s)	signer des applets
cleMail	clé privée + clé publique + certificat(s)	signer des mails
virginie	certificat	authentifier
philippe	certificat	authentifier
roland	certificat	authentifier

La manière dont un objet KeyStore est effectivement implémenté est laissée à la discréption des providers. Une implémentation par défaut est fournie par Sun : appelé "**jks**", elle utilise **un fichier au format propriétaire** (nommé classiquement **.keystore**) et est définie dans le fichier java.security sous la rubrique

keystore.type=jks.

Les providers peuvent fournir d'autres types, par exemple "**pkes12**" ou "**jceks**". On peut un objet keystore en spécifiant un type dans la méthode de classe :

```
public static KeyStore getInstance(String type) throws KeyStoreException
```

les providers potentiels étant à nouveau spécifiés dans le fichier java.security (rubrique keystore=...). Bien sûr, la classe comporte des méthodes permettant d'ajouter des entrées à l'objet créé :

```
public final void setKeyEntry(String alias, byte[] key, Certificate[] chain)
    throws KeyStoreException
```

```
public final void setKeyEntry(String alias, Key key, char[] password,
    Certificate[] chain) throws KeyStoreException
```

et

```
public final void setCertificateEntry(String alias, Certificate cert)
    throws KeyStoreException
```

On retrouve les informations dans cette "base de données" au moyen de méthodes prévisibles :

```
public final Key getKey(String alias, char[] password)
    throws KeyStoreException, NoSuchAlgorithmException, UnrecoverableKeyException
```

et

```
public final Certificate getCertificate(String alias) throws KeyStoreException
```

Cette dernière méthode fournit donc le certificat associé à l'alias précisé. A remarquer que la classe Certificate est celle du package java.security.cert (et pas java.security – cette classe-là existe déjà dans le JDK 1.1). On trouve dans cette classe Certificate la méthode

```
public abstract void verify(PublicKey key)
    throws CertificateException, NoSuchAlgorithmException, InvalidKeyException,
        NoSuchProviderException, SignatureException
```

qui permet de vérifier que le certificat a été signé au moyen de la clé privée associée à la clé publique passée en argument.

Bien clairement, on peut aussi gérer les clés au moyen du fichier keystore – un outil en ligne de commande est fourni ...

14.4 L'outil keytool

On peut aussi gérer un KeyStore au moyen d'un *outil en ligne de commande* appelé **keytool**, outil qui est apparu avec le JDK 1.2. On peut grâce à notamment lui générer un certificat signé par l'utilisateur. La paire de clés produite (option `-genkey`), selon l'algorithme précisé (dans l'option `-keyalg`) correspondra à une entrée de type "Key entry", la clé publique étant celle faisant l'objet du certificat. L'entrée en question peut être désignée par un alias (précisé dans l'option `-alias`). Le nombre de bits utilisés peut être spécifié par l'option `-keysize` (of course !). On se souviendra enfin qu'un certificat contient une identification du propriétaire. L'option `-dname` permet de définir un *distinguished name* (**DN**), qui décrit une personne de manière normalisée selon les champs (non obligatoires) :

- ◆ CN (Common Name) : le nom;
- O(Organization) : l'organisme auquel la personne appartient;
- ◆ OU (Organization Unit) : le service auquel cette personne appartient;
- ◆ L (Locality) : la ville;
- ◆ S (State) : l'état au sens américain, donc en Europe la province, le département, le county;
- ◆ C (Country) : le pays.

On peut donc imaginer ceci :

```
C:\jdk12\bin>keytool -genkey -alias Claude -keyalg DSA -keysize 1024 -dname "CN=Claude Vilvens, O=HEP R Sualem, C=B"
```

ce qui donne comme réponse

```
Enter keystore password: beaugosse  
Enter key password for <Claude>  
      (RETURN if same as keystore password):
```

Le deuxième mot de passe sert à protéger la clé privée. Le premier sert à protéger l'accès au KeyStore lui-même. Celui-ci a été créé dans un fichier **.keystore** qui, par défaut, a été placé, par exemple, dans le répertoire Windows ou le répertoire vilvens (nom de l'utilisateur d'après la propriété user.name du système). On aurait pu créer un fichier particulier avec le commutateur `-keystore` suivi du nom du fichier. Les deux mots de passe auraient pu être précisés sur la ligne de commande par les commutateurs `-storepass` et `-keypass`.

On peut vérifier l'entrée créée par :

```
C:\jdk12\bin>keytool -list  
Enter keystore password: beaugosse
```

ce qui donne :

```
Keystore type: jks  
Keystore provider: SUN
```

Your keystore contains 1 entry:
claudie, Fri May 18 18:39:18 CEST 2001, keyEntry,
Certificate fingerprint (MD5): 09:23:BE:FC:63:0A:F8:F8:C7:08:DB:E3:1F:8C:20:63

Par " *Certificate fingerprint*" , il faut comprendre un digest-signature qui a été calculé sur le certificat entier. La version plus "verbeuse" (avec le commutateur **-v**) donne :

```
Keystore type: jks
Keystore provider: SUN

Your keystore contains 1 entry:

Alias name: claudie
Creation date: Fri May 18 18:39:18 CEST 2001
Entry type: keyEntry
Certificate chain length: 1
Certificate[1]:
Owner: CN=Claude Vilvens, O=HEP R Sualem, C=B
Issuer: CN=Claude Vilvens, O=HEP R Sualem, C=B
Serial number: 3b05501c
Valid from: Fri May 18 18:38:52 CEST 2001 until: Thu Aug 16 18:38:52 CEST 2001
Certificate fingerprints:
    MD5: 09:23:BE:FC:63:0A:F8:F8:C7:08:DB:E3:1F:8C:20:63
    SHA1: 72:AB:CD:E1:95:D8:DF:1C:34:2D:56:93:71:EF:07:5D:69:1A:DF:C7
```

14.5 Créeer un fichier certificat

Il est possible de placer un certificat précis dans un fichier désigné. Ainsi, pour notre exemple :

```
C:\jdk12\bin>keytool -export -alias claudie -file james.cer
Enter keystore password: beaugosse
Certificate stored in file <james.cer>
```

On peut vérifier ce que contient le fichier :

```
C:\jdk12\bin>keytool -printcert -file james.cer
Owner: CN=Claude Vilvens, O=HEP R Sualem, C=B
Issuer: CN=Claude Vilvens, O=HEP R Sualem, C=B
Serial number: 3b05501c
Valid from: Fri May 18 18:38:52 CEST 2001 until: Thu Aug 16 18:38:52 CEST 2001
Certificate fingerprints:
    MD5: 09:23:BE:FC:63:0A:F8:F8:C7:08:DB:E3:1F:8C:20:63
    SHA1: 72:AB:CD:E1:95:D8:DF:1C:34:2D:56:93:71:EF:07:5D:69:1A:DF:C7
```

Remarque

Rien n'empêche de placer plusieurs certificats dans le keystore :

```
C:\jdk12\bin>keytool -genkey -alias Denys -keyalg DSA -keysize 1024 -dname "CN=Denys
Mercenier, O=HEP R Sualem,L="Seraing", C=B"
Enter keystore password: beaugosse
Enter key password for <Denys>
(RETURN if same as keystore password): beaumec
```

Le mot de passe est différent pour le nouvel alias. La liste donne :

```
C:\jdk12\bin>keytool -list  
Enter keystore password: beaugosse  
Keystore type: jks  
Keystore provider: SUN  
Your keystore contains 2 entries:  
claudie, Fri May 18 18:39:18 CEST 2001, keyEntry,  
Certificate fingerprint (MD5): 09:23:BE:FC:63:0A:F8:F8:C7:08:DB:E3:1F:8C:20:63  
denys, Mon May 21 17:45:06 CEST 2001, keyEntry,  
Certificate fingerprint (MD5): 7B:B4:F9:E2:C0:D3:33:61:F2:FE:A1:06:C6:C0:D3:02
```

Une entrée peut toujours être détruite avec le commutateur **–delete** suivi de la spécification de l'alias (par **–alias**).

14.6 Obténir un certificat

Le certificat généré ci-dessus est un **certificat "auto-signé"** (*self-signed certificate*), qui contient la clé publique correspondant à la clé privée; en fait, le signataire d'un tel certificat est le même que celui dont la clé publique fait l'objet de la certification.

Pour obtenir un **certificat réel**, autre que celui généré ci-dessus et qui attestera de sa clé publique, un utilisateur quelconque doit construire une demande que l'on appelle un **CSR** (Certificate Signing Request). Il s'agit d'un fichier qui contient

- ◆ la clé publique du demandeur;
- ◆ la signature construite au moyen de la clé privée du demandeur.

On génère un tel CSR au moyen de l'utilitaire **keytool**, , principalement orné de la directive **–certreq** (ou **–csr**) :

```
C:\jdk12\bin>keytool -certreq -alias claudie -file claudie.csr -keystore beaugosse  
-storepass beaugosse -v  
Certification request stored in file <claudie.csr>  
Submit this to your CA
```

On aura compris que la clause **–file** permet de désigner le fichier qui contiendra la requête. Ce fichier est créé dans le répertoire courant. Dans notre cas, il contient (en format PEM) :

```
-----BEGIN NEW CERTIFICATE REQUEST-----  
MIICQTCCAf8CAQAwPDEKMAgGA1UEBhMBQjEVMBMGA1UEChMMSEVQIFIgU3VhbGVtMRcwFQYDVQ  
QDEw5DbGF1ZGUgVmlsdmVuczCCAAbgwggEsBgcqhkjOOAQBMIIHBwKBgQD9f1OBHXUSKVLfSpwu7OTn  
9hG3UjzvRADDHj+A1lEmaUVdQCJR+1k9jVj6v8X1ujD2y5tVbNeBO4AdNG/yZmC3a5lQpaSfn+gEexAiwk+7qdf+t  
8Yb+DtX58aophUPBPuD9tPFHsMCNVQTWhaRMvZ1864rYdcq7/IiAxmd0UgBxwIVAJdgUI8VIwvMspK5gqLrhA  
vwWBz1AoGBAPfhoIXWmz3ey7yrXDa4V7I5IK+7+jrqgvIXTAs9B4JnUVIXjrrUWU/mcQcQgYC0SRZxI+hMKBY  
Tt88JMozIpue8FnqLVHyNKOcjrh4rs6Z1kW6jfww6ITVi8ftiegEkO8yk8b6oUZCJqIPf4VrlnwSi2ZegHtVJWQBTd  
+z0kqA4GFAAKBgQCHhd+c6vgg0TM4oL7WhDQ5cMinHHZeBwlzfV76sSneFuOf6zqDDfaEvzyUFKqdoc1LXJD  
DiNIH4Qx0Wmqz2JzEEPIPhMG1G10V7BbHKt6i2INvzSD55p4ca/fx36OG2r930a39f9WNj3BnsAGIEQmfpxmjT  
CtPOAijpev962aKAAMAsGBYqGSM44BAMFAAMvADAsAhQcppu1OF5Cjh3R0p/HvE2p8CI86wIUEZb72XFat5D  
wK/Pwp7oxcebRv+o=  
-----END NEW CERTIFICATE REQUEST-----
```

Le CA, après réception, pourra alors vérifier le bien fondé de la demande et, dans l'affirmative, fournir le certificat demandé. Celui-ci, tout comme la requête d'ailleurs, est mémorisé dans un format d'encodage imprimable défini par la RFC 1421, format appelé le format **Base64** et particulièrement utilisable par les applications e-mail (fichiers **.pem**). Ce format est en fait un système de représentation de tableaux de bytes en caractères ASCII et utilise 6 bits par chiffre (tout comme l'hexadécimal en utilise 4). Le certificat a le look suivant, obtenu avec keytool muni du commutateur –printcert) :

```
-----BEGIN CERTIFICATE -----  
MIICMTCCAZoCAS...  
..  
..  
...  
=====  
-----END CERTIFICATE -----
```

Ce certificat, intimement associé à la clé privée qui a servi à signer le CSR, sera stocké dans le keystore au moyen de la commande d'importation keytool –import :

```
C:\jdk12\bin>keytool -import -alias claude -file jcrtjames.cer  
Enter keystore password: beaugosse
```

Dans ce cas où l'alias correspond à une entrée dans le keystore, le certificat initial, auto-signé, sera ainsi remplacé par une **chaîne de certificats**. A la fin de cette chaîne se trouvent le certificat reçu en réponse du CA et le certificat qui authentifie la clé publique de ce CA. Le plus souvent, il s'agit d'une certificat auto-signé par lui mais il pourrait aussi s'agir d'un certificat signé par un autre CA. On voit évidemment comment la chaîne pourrait se poursuivre ... jusqu'à parvenir à un CA suffisamment connu pour que sa clé publique soit connue par des publications ou tout autre moyen digne de foi.

14.7 **Créer une entrée pour un certificat sûr**

Dans le cas où l'on reçoit un certificat d'un tiers (qui n'a donc pas d'alias dans le Keystore), on créera une "Trusted Certificate Entry". Par exemple, si l'on a créé un autre keystore (le fichier est précisé au moyen de l'option **-keystore**) par :

```
C:\jdk12\bin>keytool -genkey -alias Denys -keyalg DSA -keysize 1024 -dname "CN=D  
enys Mercenier, O=HEP R Sualem,L="Seraing", C=B" -keystore secret.key  
Enter keystore password: supernana  
Enter key password for <Denys>  
(RETURN if same as keystore password): supergirl
```

on peut y importer un certificat par :

```
C:\jdk12\bin>keytool -import -keystore secret.key -alias claude -file james.cer  
Enter keystore password: supernana  
Owner: CN=Claude Vilvens, O=HEP R Sualem, C=B  
Issuer: CN=Claude Vilvens, O=HEP R Sualem, C=B  
Serial number: 3b05501c  
Valid from: Fri May 18 18:38:52 CEST 2001 until: Thu Aug 16 18:38:52 CEST 2001  
Certificate fingerprints:  
MD5: 09:23:BE:FC:63:0A:F8:F8:C7:08:DB:E3:1F:8C:20:63
```

```
SHA1: 72:AB:CD:E1:95:D8:DF:1C:34:2D:56:93:71:EF:07:5D:69:1A:DF:C7
Trust this certificate? [no]: y
Certificate was added to keystore
```

On peut vérifier le contenu du keystore par :

```
C:\jdk12\bin>keytool -list -keystore secret.key
Enter keystore password: supernana
Keystore type: jks
Keystore provider: SUN
Your keystore contains 2 entries:
claudie, Tue May 22 17:36:48 CEST 2001, trustedCertEntry,
Certificate fingerprint (MD5): 09:23:BE:FC:63:0A:F8:F8:C7:08:DB:E3:1F:8C:20:63
denys, Tue May 22 17:32:08 CEST 2001, keyEntry,
Certificate fingerprint (MD5): 96:7C:88:32:58:B4:7A:D9:52:D9:C9:82:F8:9A:F3:D0
```

Tout cela est très bien. Mais nous ne savons toujours pas comment un certificat s'utilise en pratique ... Voici donc deux applications.

14.8 Les classes certificats en Java

Nous y avons déjà fait allusion, Java fournit des APIs permettant de manipuler les certificats au sein d'une application. De la classe basique **Certificate**, une classe **X509Certificate** a été dérivée au sein du package `java.security.cert`. Avec un nom pareil, on se doute qu'elle correspond aux certificats codés selon cette norme. En fait, il s'agit simplement d'une classe de référence qui ne comporte que des méthodes abstraites. On en obtiendra une implémentation utilisable en utilisant la classe factory **CertificateFactory**, dont on obtient une instance avec la méthode factory :

```
public static final CertificateFactory getInstance(String type)
    throws CertificateException
```

Supposons donc que nous disposions d'un certificat créé selon les normes d'encodage DER (Distinguished Encoding Rule) qui sont celles utilisées dans les certificats X509 :



On obtient une représentation mémoire du certificat se trouvant dans ce fichier **.der** en utilisant la méthode de la classe **CertificateFactory**:

```
public final Certificate generateCertificate(InputStream inStream)
    throws CertificateException
```

à laquelle il suffit de passer un objet `FileInputStream` construit dessus. Il n'y a dès lors plus qu'à extraire de l'objet certificat tous les renseignements désirés au moyen de méthodes comme :

```
public abstract Date getNotAfter()
```

```
public abstract byte[] getSignature()
public abstract Principal getIssuerDN()
etc
```

la classe Principal n'étant jamais qu'une entité qui encapsule un nom que l'on obtient par :

```
public String getName()
```

Donc finalement :

Certificats.java

```
import java.io.*;
import java.security.cert.*;
import java.security.*;

public class Certificats
{
    public static void main(String[] args)
    {
        InputStream inStream = null;
        try
        {
            inStream = new FileInputStream(
                "C:\\java-sun-application\\Certificates\\clientJsse.der");
        }
        catch (FileNotFoundException e)
        {
            System.out.println("Fichier certificat : " + e.getMessage());
        }

        CertificateFactory cf = null;
        X509Certificate cert = null;
        try
        {
            cf = CertificateFactory.getInstance("X.509");
            cert = (X509Certificate)cf.generateCertificate(inStream);

            System.out.println("Classe instanciée : " + cert.getClass().getName());
            System.out.println("Type de certificat : " + cert.getType());
            System.out.println("Nom du propriétaire du certificat : " +
                cert.getSubjectDN().getName());
            PublicKey clePublique = cert.getPublicKey();
            System.out.println("... sa clé publique : " + clePublique.toString());
            System.out.println("... la classe instanciée par celle-ci : " +
                clePublique.getClass().getName());
            System.out.println("Dates limites de validité : [" + cert.getNotBefore() + " - " +
                cert.getNotAfter() + "]");

            System.out.println("Signataire du certificat : " + cert.getIssuerDN().getName());
        }
```

```
System.out.println("Algo de signature : " + cert.getSigAlgName());
System.out.println("Signature : " + cert.getSignature());
}
catch (CertificateException e)
{
    System.out.println("Certificat : " + e.getMessage());
}

try { inStream.close(); }
catch (FileNotFoundException e)
{ System.out.println("Fichier certificat : " + e.getMessage()); }
catch (IOException e)
{ System.out.println("Fichier certificat : " + e.getMessage()); }

try
{
    cert.checkValidity();
}
catch (CertificateExpiredException e)
{
    System.out.println("Certificat périmé : " + e.getMessage());
}
catch (CertificateNotYetValidException e)
{
    System.out.println("Certificat pas encore valide : " + e.getMessage());
}
}
```

Un exemple d'exécution pourrait être :

Classe instanciée : **sun.security.x509.X509CertImpl**

Type de certificat : **X.509**

Nom du propriétaire du certificat : CN=Claude Vilvens, OU=Dept. Informatique (clients),
O=HEP R Sualem, ST=Liège, C=BE

... sa clé publique : Sun DSA Public Key

Parameters:DSA

p: fd7f5381 1d751229 52df4a9c 2eece4e7 f611b752 3cef4400 c31e3f80 b6512669
455d4022 51fb593d 8d58fabf c5f5ba30 f6cb9b55 6cd7813b 801d346f f26660b7
6b9950a5 a49f9fe8 047b1022 c24fbba9 d7feb7c6 1bf83b57 e7c6a8a6 150f04fb
83f6d3c5 1ec30235 54135a16 9132f675 f3ae2b61 d72aeff2 2203199d d14801c7

q: 9760508f 15230bcc b292b982 a2eb840b f0581cf5

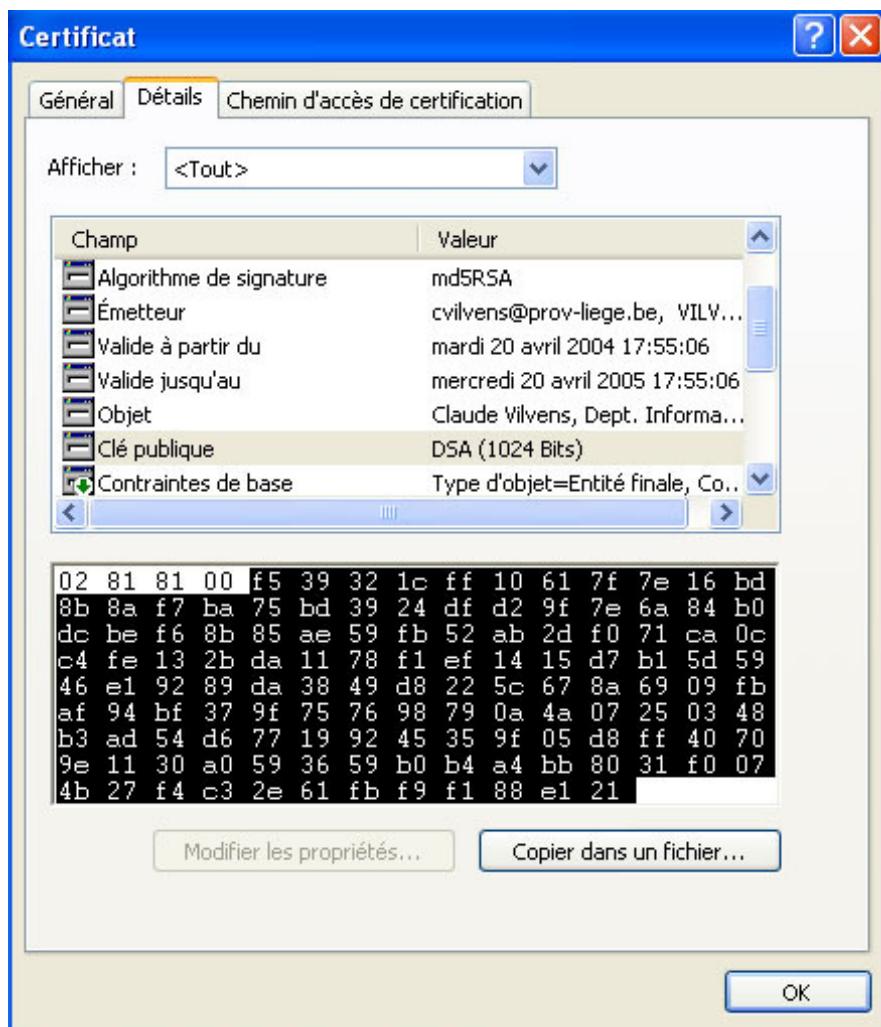
g: f7e1a085 d69b3dde cbbcab5c 36b857b9 7994afbb fa3aea82 f9574c0b 3d078267
5159578e bad4594f e6710710 8180b449 167123e8 4c281613 b7cf0932 8cc8a6e1
3c167a8b 547c8d28 e0a3ae1e 2bb3a675 916ea37f 0bfa2135 62f1fb62 7a01243b
cca4f1be a8519089 a883dfe1 5ae59f06 928b665e 807b5525 64014c3b fecf492a

y:

f539321c ff10617f 7e16bd8b 8af7ba75 bd3924df d29f7e6a 84b0dcbe f68b85ae
59fb52ab 2df071ca 0cc4fe13 2bda1178 f1ef1415 d7b15d59 46e19289 da3849d8
225c678a 6909fbaf 94bf379f 75769879 0a4a0725 0348b3ad 54d67719 9245359f

05d8ff40 709e1130 a0593659 b0b4a4bb 8031f007 4b27f4c3 2e61fbf9 f188e121
... la classe instanciée par celle-ci : sun.security.provider.DSAPublicKey
Dates limites de validité : [Tue Apr 20 17:55:06 CEST 2004 - Wed Apr 20 17:55:06 CEST
2005]
Signataire du certificat : EMAILADDRESS=cvilvens@prov-liege.be, CN=" VILVENS
Claude", OU=Dept. Informatique, O=HEP R Sualem, L=Oupeye, ST=Liege, C=BE
Algo de signature : MD5withRSA
Signature : [B@b2fd8f
Certificat périmé : NotAfter: Wed Apr 20 17:55:06 CEST 2005

A comparer avec ce que lit Internet Explorer :



15. L'outil Keytool IUI

15.1 Présentation et installation

L'outil graphique Keytool IUI est une application GUI écrite en Java, disponible sur <http://yellowcat1.free.fr/>, qui permet de gérer bon nombre d'objets de cryptographie de manière plus aisée et plus puissante que l'outil keytool en ligne de commandes faisant partie du JDK. L'outil réclame un JDK 1.6 et utilise la librairie BouncyCastle. Une fois le fichier ktl241sta.rar décompressé, on obtient les jars de la librairie (y compris ceux de BouncyCastle) ainsi qu'un fichier run_ctl.bat destiné à lancer l'application. Il faut simplement au préalable éditer ce fichier pour régler la valeur de la variable d'environnement HOME_JAVA qui contient le chemin exact de l'interpréteur Java :

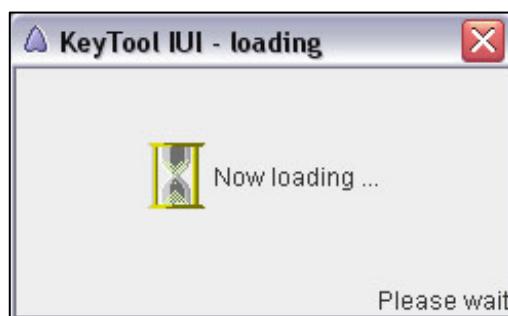
run_ctl.bat

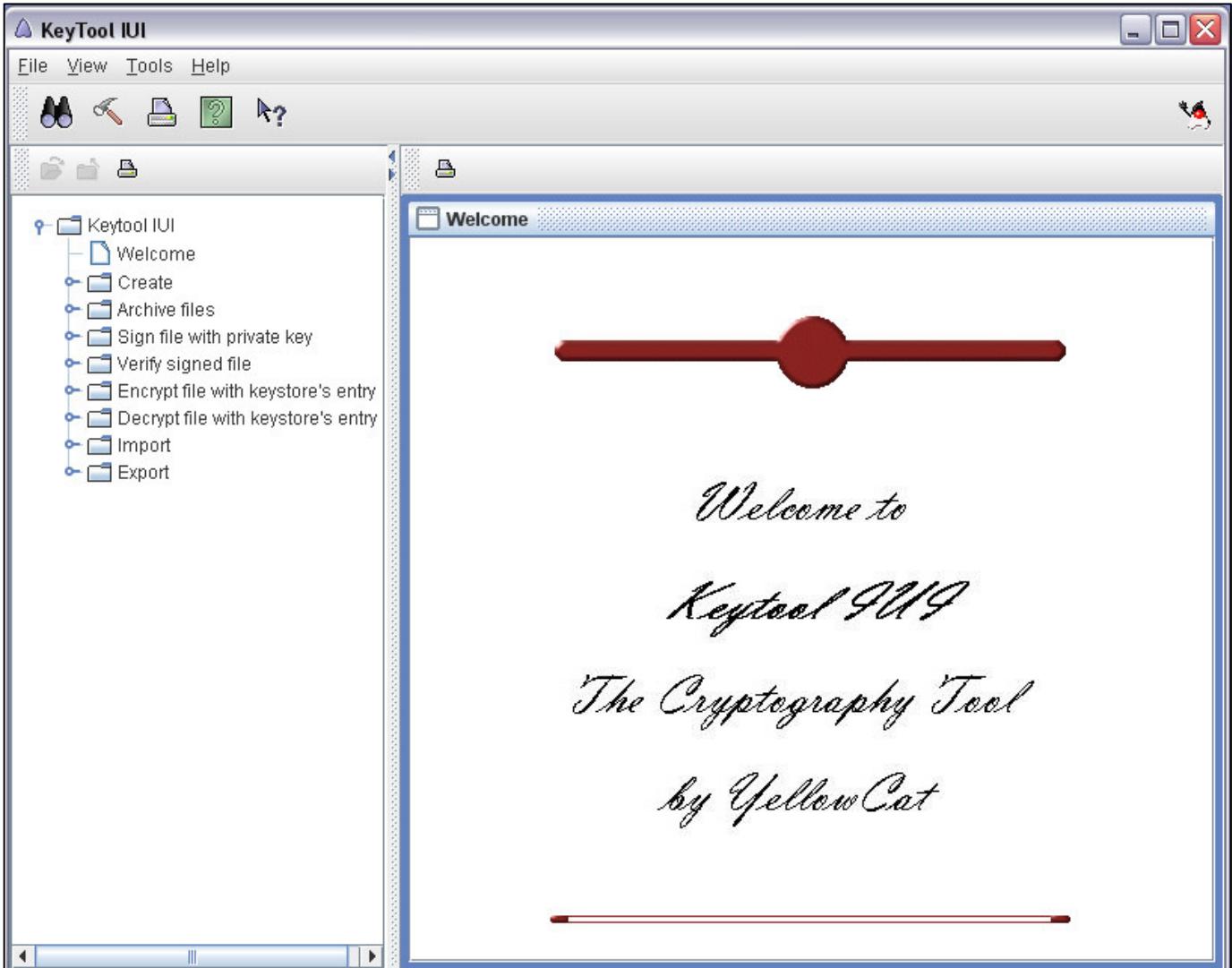
```
@echo off  
cls  
  
REM --- BEG MACHINE-DEPENDENT ----  
REM set HOME_JAVA=jav  
set HOME_JAVA=C:\Program Files\Java\jdk1.6.0_10\jre\bin\java  
REM --- END MACHINE-DEPENDENT ----  
  
set HOME_JAVA="%HOME_JAVA%"  
set ARG_MEMORY=-Xms128m -Xmx196m  
set MAIN_JAR=rc15ktl.jar  
  
set _CMD_= "%HOME_JAVA% %ARG_MEMORY% -jar %MAIN_JAR%"  
  
@echo %_CMD_%  
%_CMD_%
```

On peut alors lancer le batch :

```
C:\>cd C:\Keytool IUI\ktl241sta\ktl241sta  
C:\Keytool IUI\ktl241sta\ktl241sta>run_ctl  
"C:\Program Files\Java\jdk1.6.0_10\jre\bin\java" -Xms128m -Xmx196m -jar rc15ktl.  
jar
```

qui provoque l'exécution de l'application :



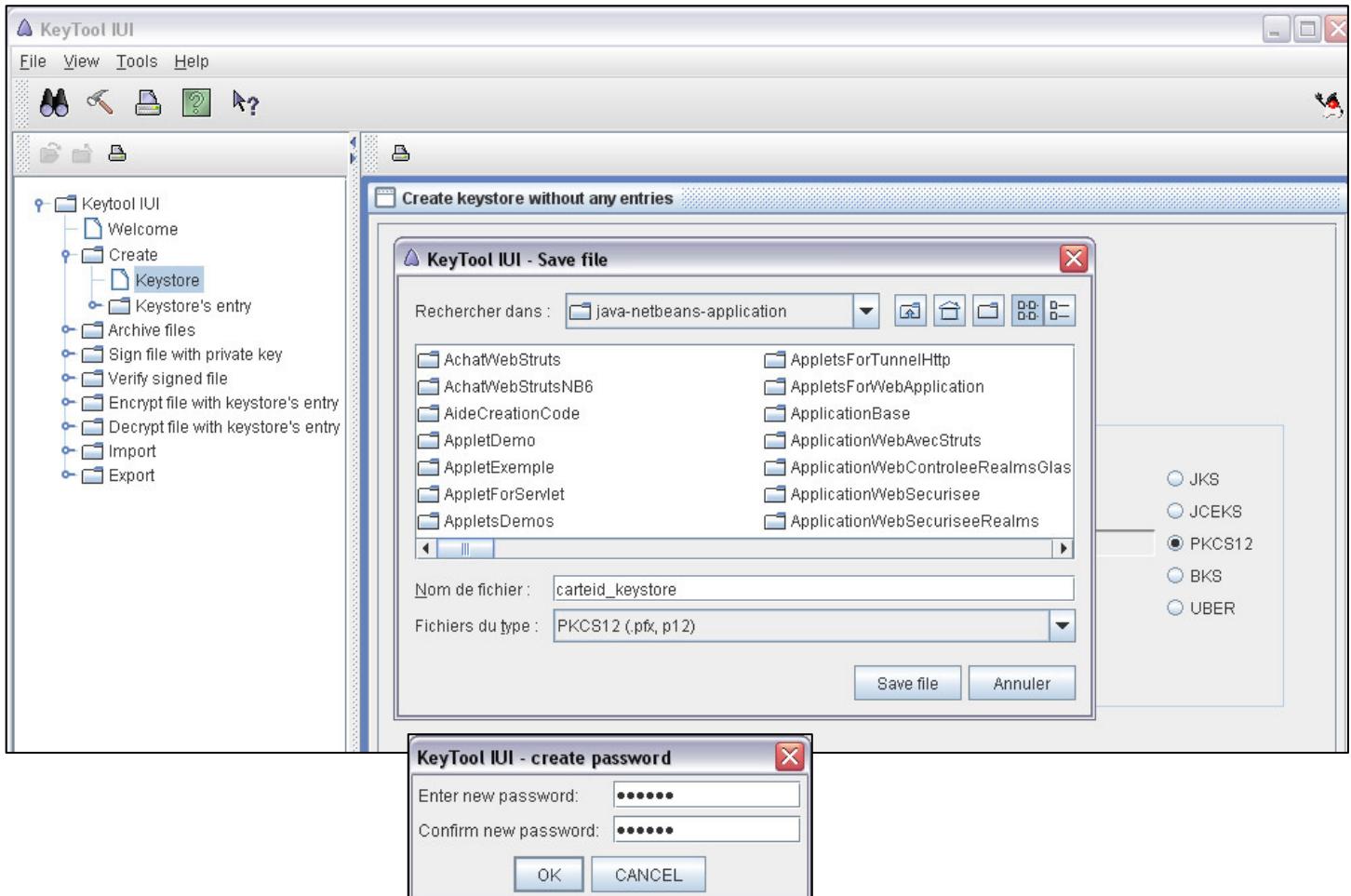


On s'aperçoit du nombre impressionnant de possibilités ...

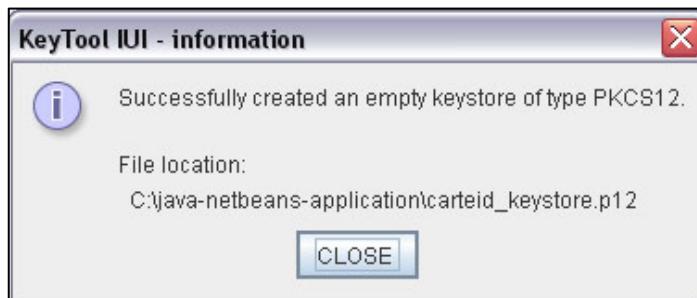
15.2 La création d'un keystore

On peut tout d'abord créer un keystore vide. Cinq formats de keystore sont disponibles :

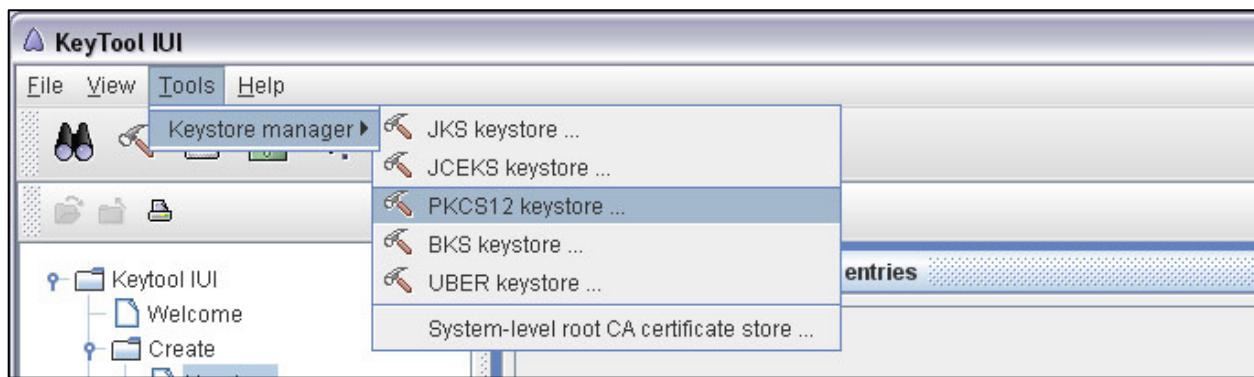
- ◆ JKS : l'implémentation standard de Sun, avec chaque entrée de clé privée seulement accessible sur production d'un second mot de passe spécifique;
- ◆ JCEKS : une amélioration du précédent, qui peut contenir des clés secrètes (symétriques);
- ◆ PKCS12 : l'implémentation Public-Key Cryptography Standards de RSA (elle est encore désignée sous le nom de "Personal Information Exchange Syntax Standard"); un seul mot de passe global est nécessaire;
- ◆ BKS : l'implémentation de Bouncy Castle, similaire au JCEKS;
- ◆ UBER : une amélioration du précédent, avec une utilisation renforcée du mot de passe.



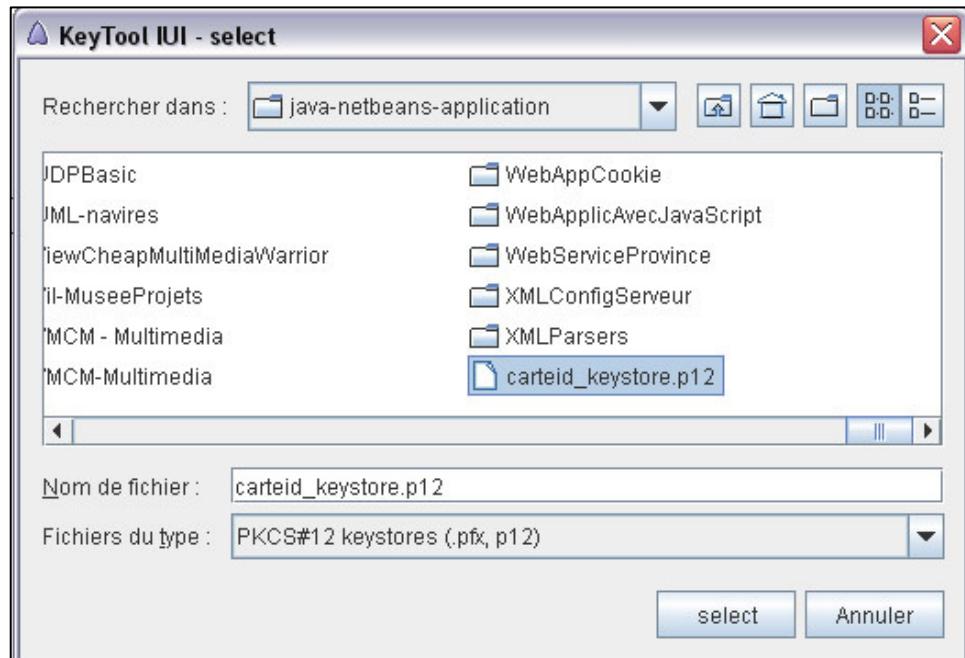
L'appui sur le bouton-icône OK donne :



On peut visualiser un keytool quelconque en utilisant l'item de menu :



qui permet de choisir le keystore désiré :

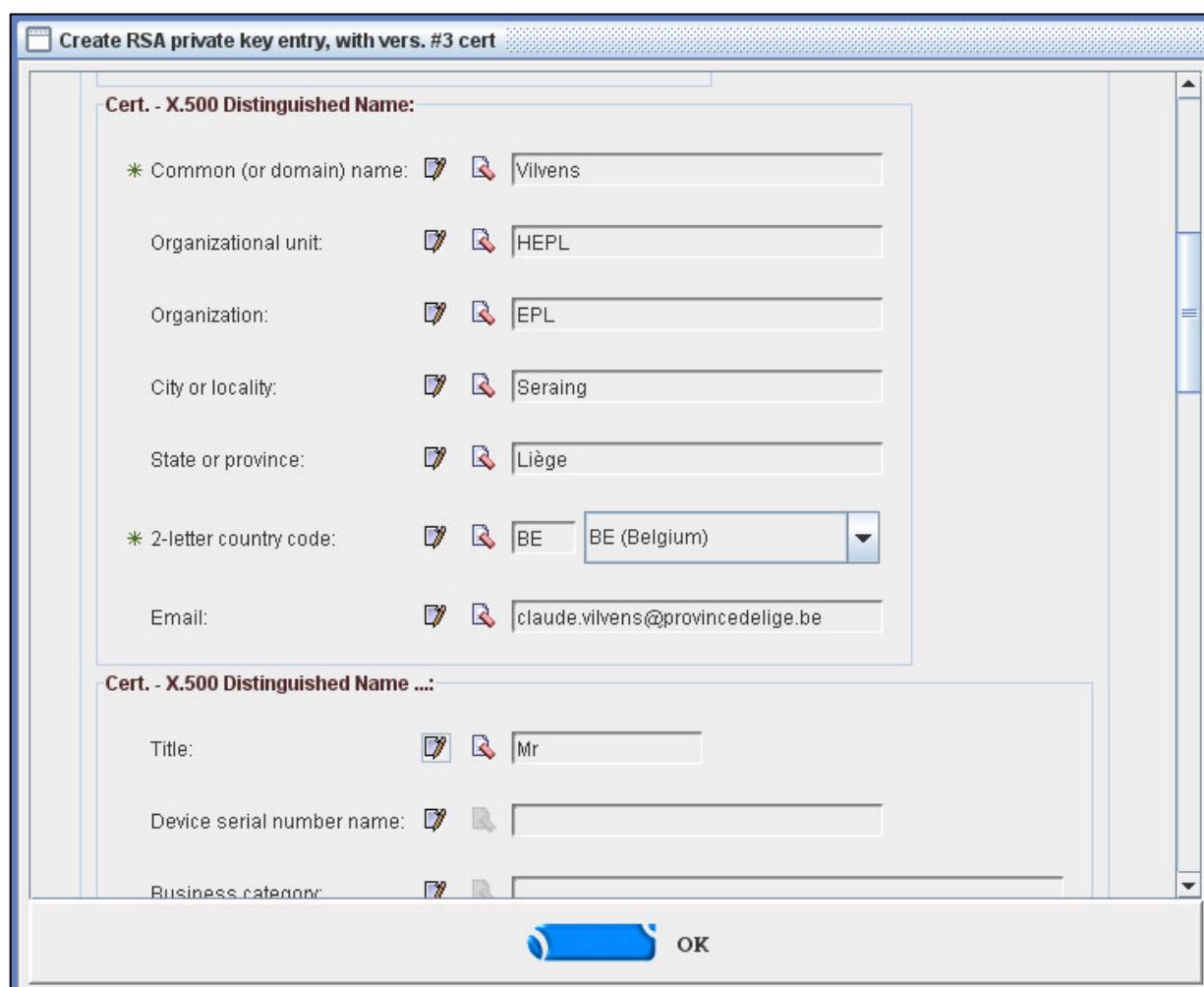
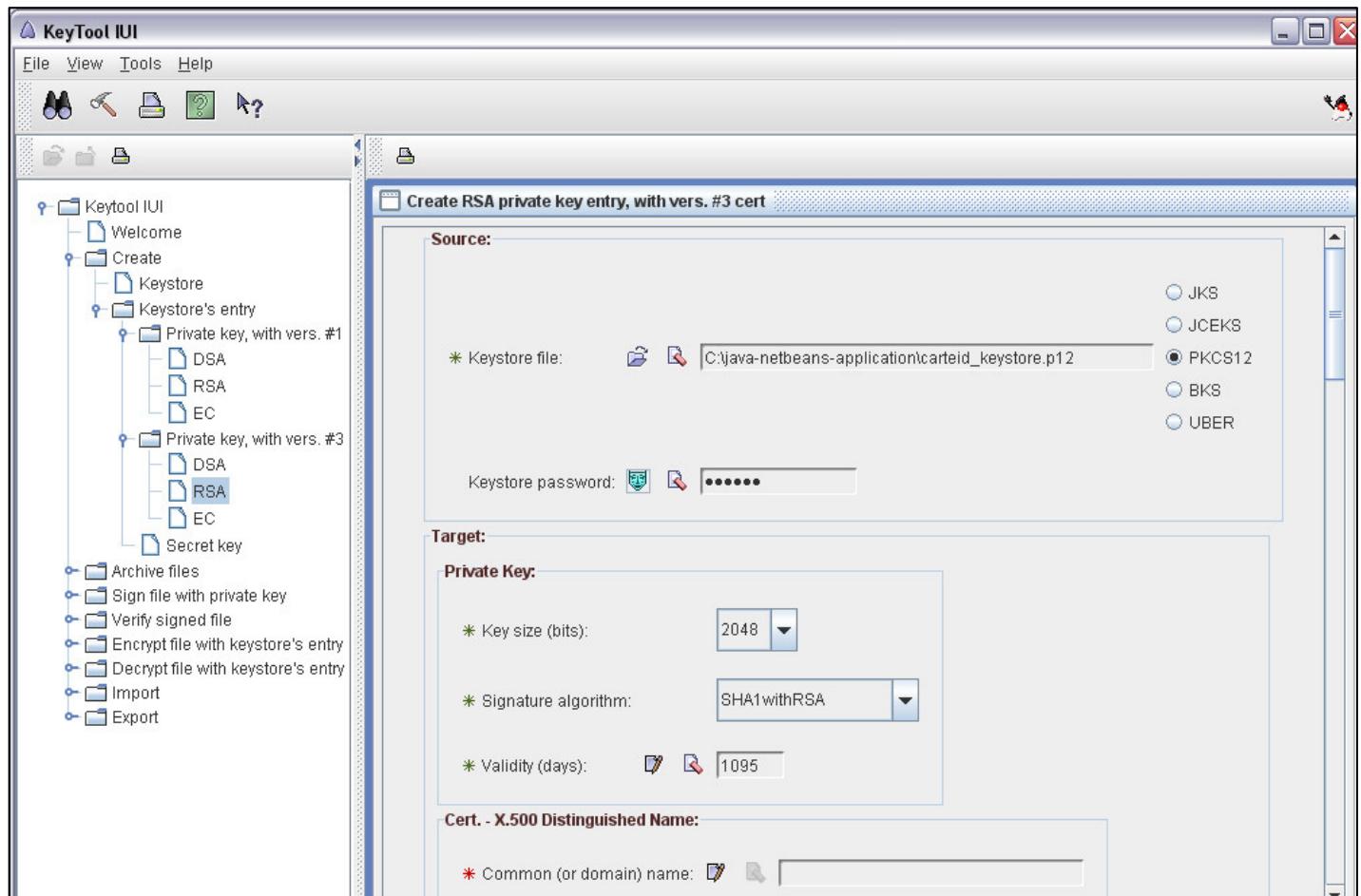


Evidemment, pour le moment, il est vide :



15.3 La génération d'une paire de clés dans un keystore PKCS12

Nous allons à présent créer une Key entry dans notre keystore, donc générer une paire de clés privée-publique. Bien sûr, on nous demandera les informations nécessaires pour générer le certificat qui contiendra la clé publique :



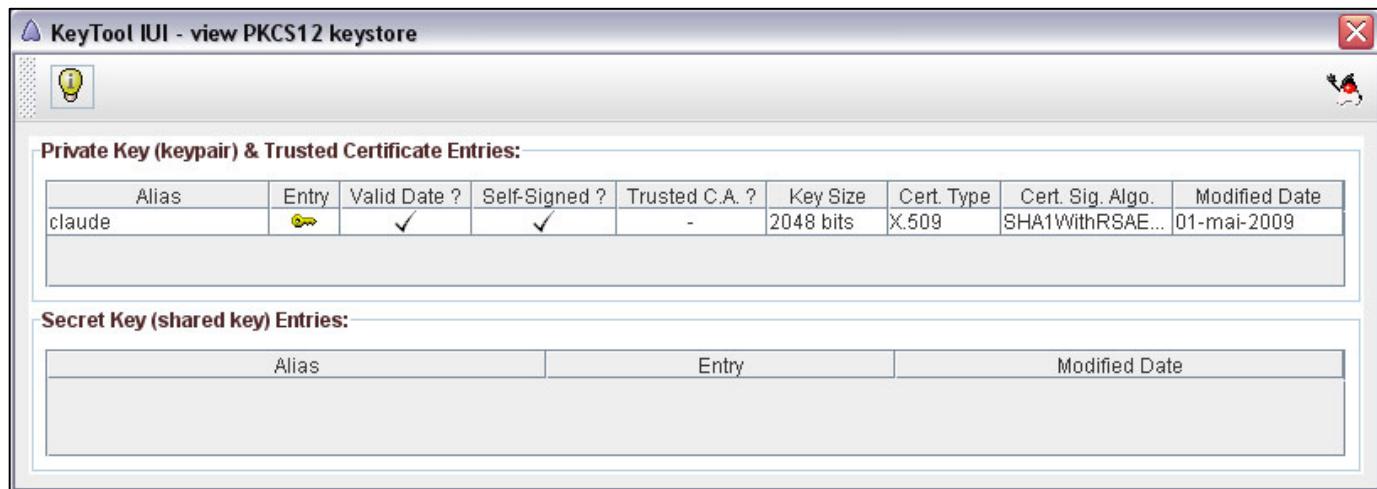
Avant de lancer la fabrication, il faudra encore donner l'alias de l'entrée du keystore :



ce qui nous donne :



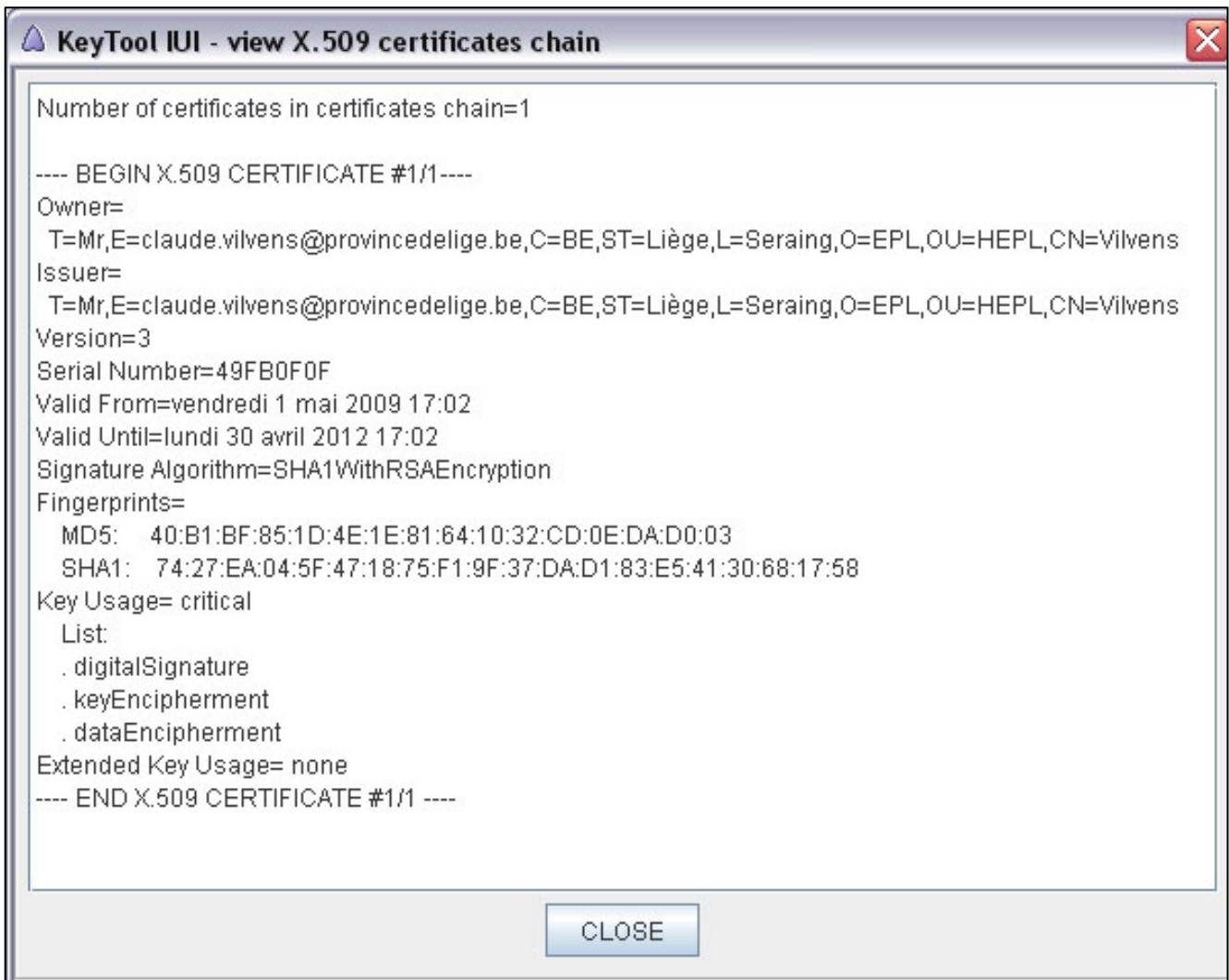
et effectivement :



Un clic droit sur l'entrée :

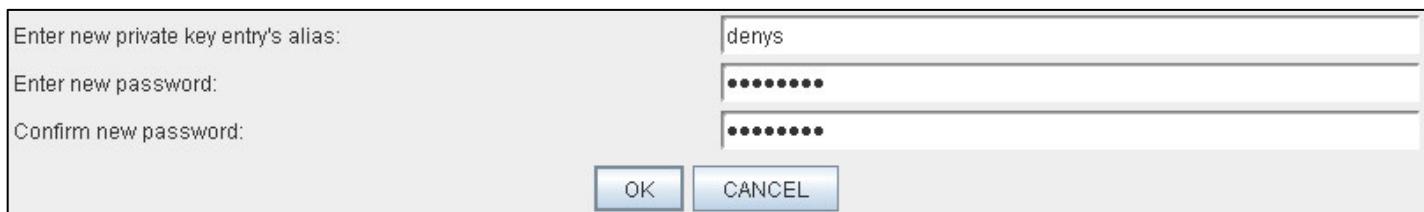
[View certificates chain ...](#)

permet d'obtenir une visualisation du certificat :



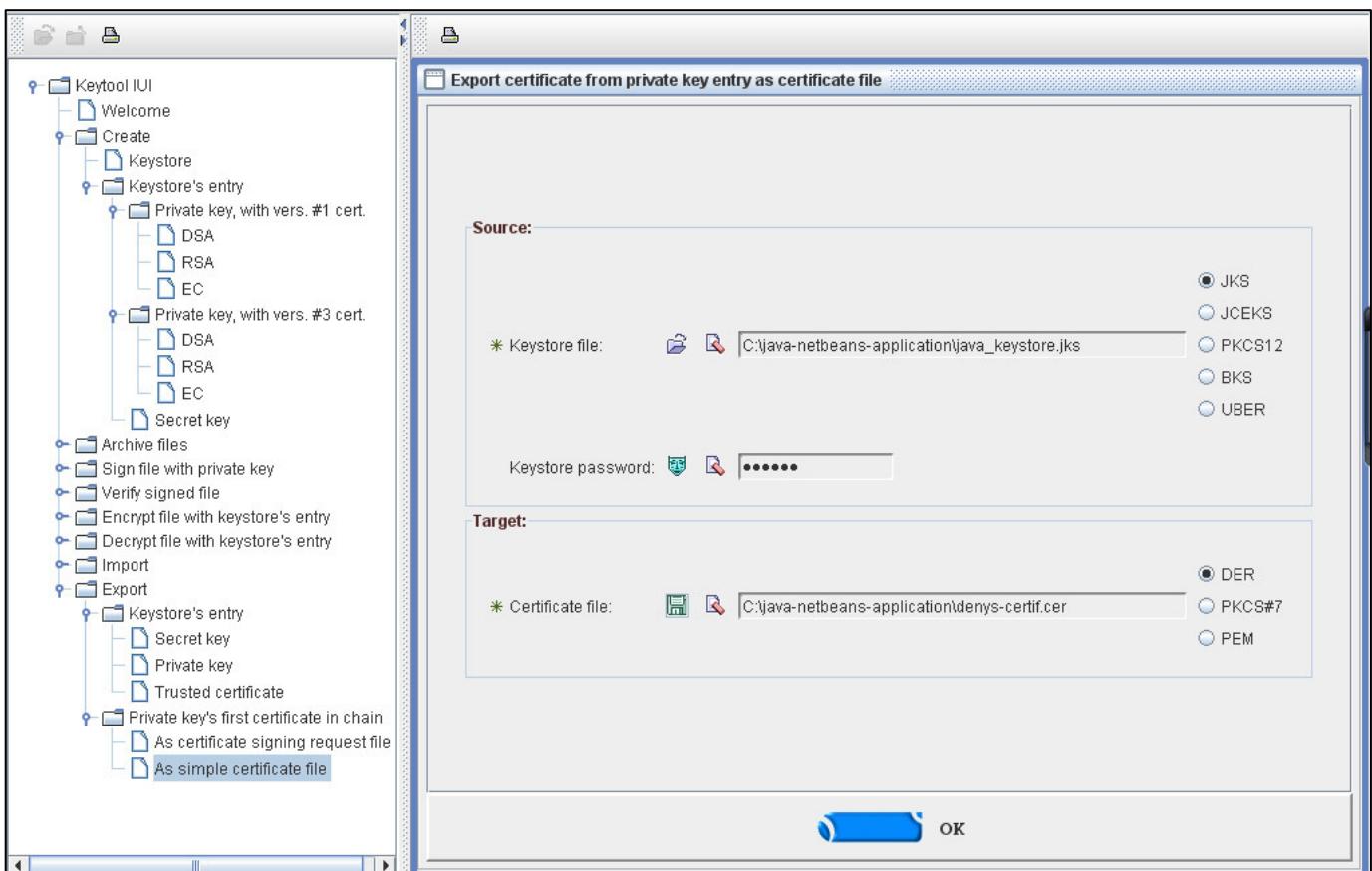
15.4 La génération d'une paire de clés dans un keystore JKS

Créons à présent un keystore implémenté au format JKS (appelons-le "java_keystore"). Tout se passe de la même manière, mais il nous sera demandé, en sus de l'alias, un mot de passe pour cette entrée :



15.5 L'exportation d'un certificat

Nous allons à présent exporter le certificat associé au couple clé privée-clé publique que nous venons de créer dans le keystore JKS :

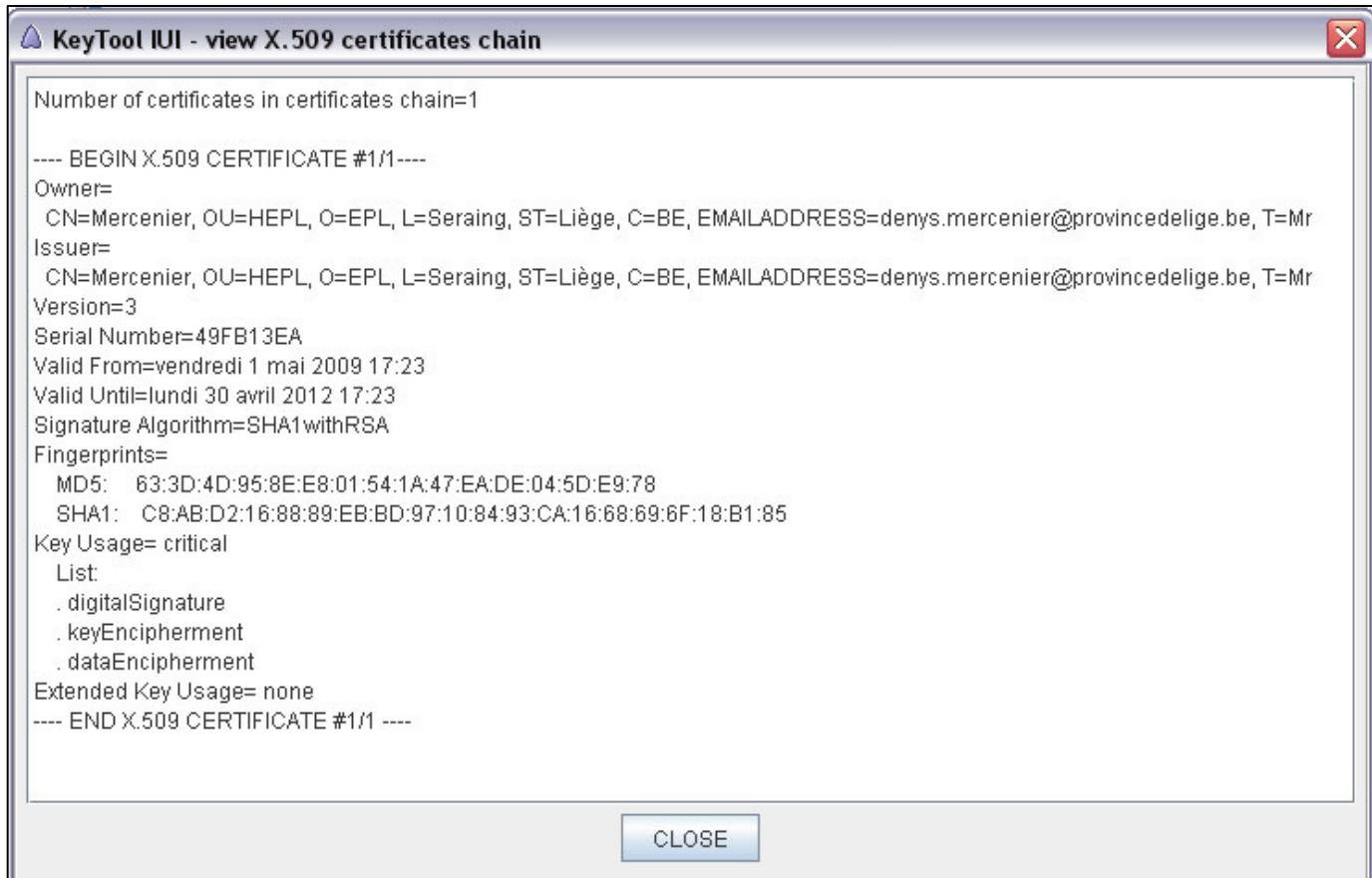


On peut voir que le certificat sera fourni sous forme d'un fichier **cer** : il s'agit d'un fichier parfaitement lisible au format X.509 qui contient les informations nécessaires pour identifier un utilisateur ou une entité (un serveur web par exemple). L'idée est qu'un utilisateur pourra vérifier la chaîne de certification et donc la validité du certificat. Bien logiquement, il faut choisir l'entrée concernée :





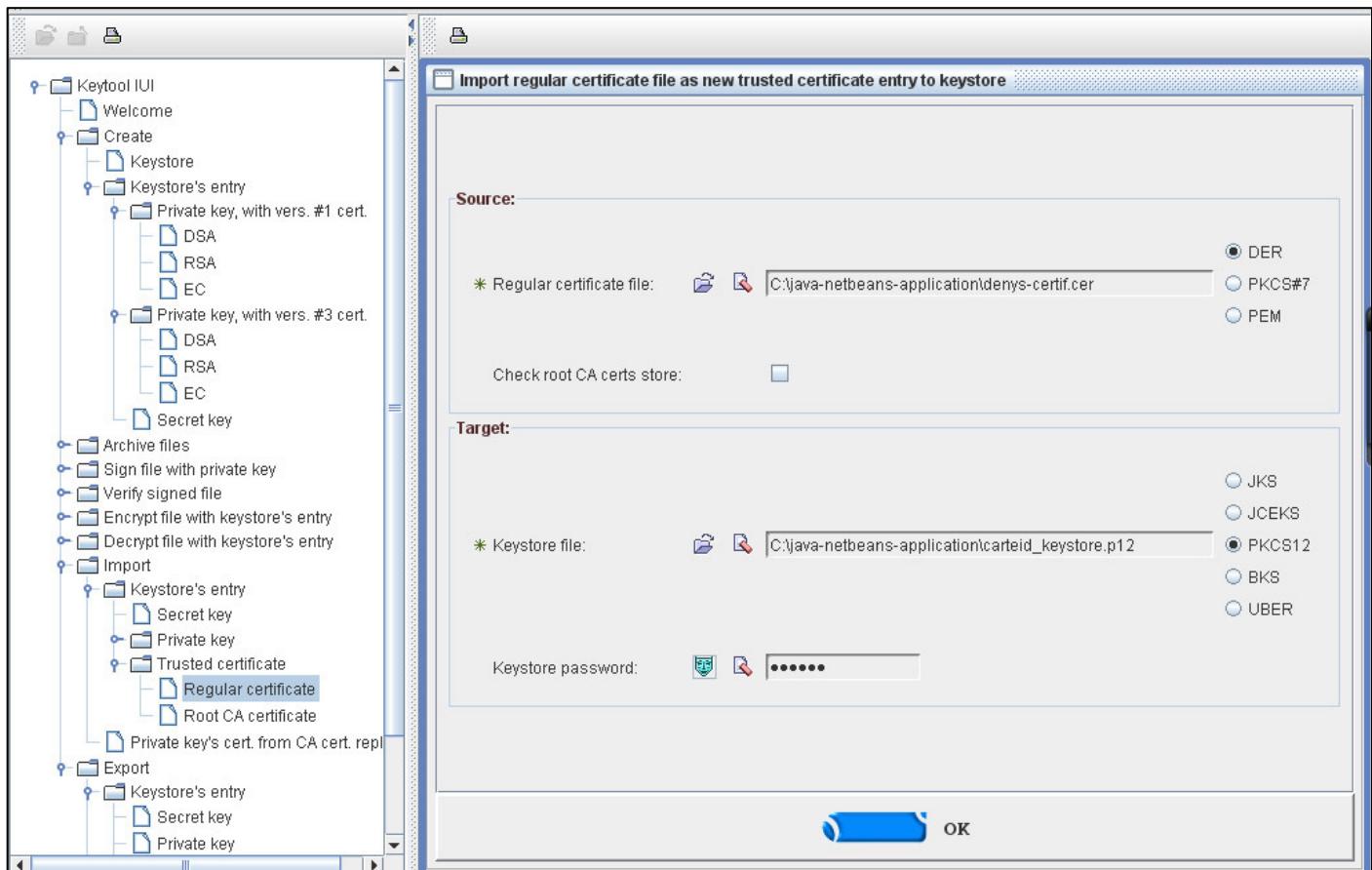
C'est bien ça :



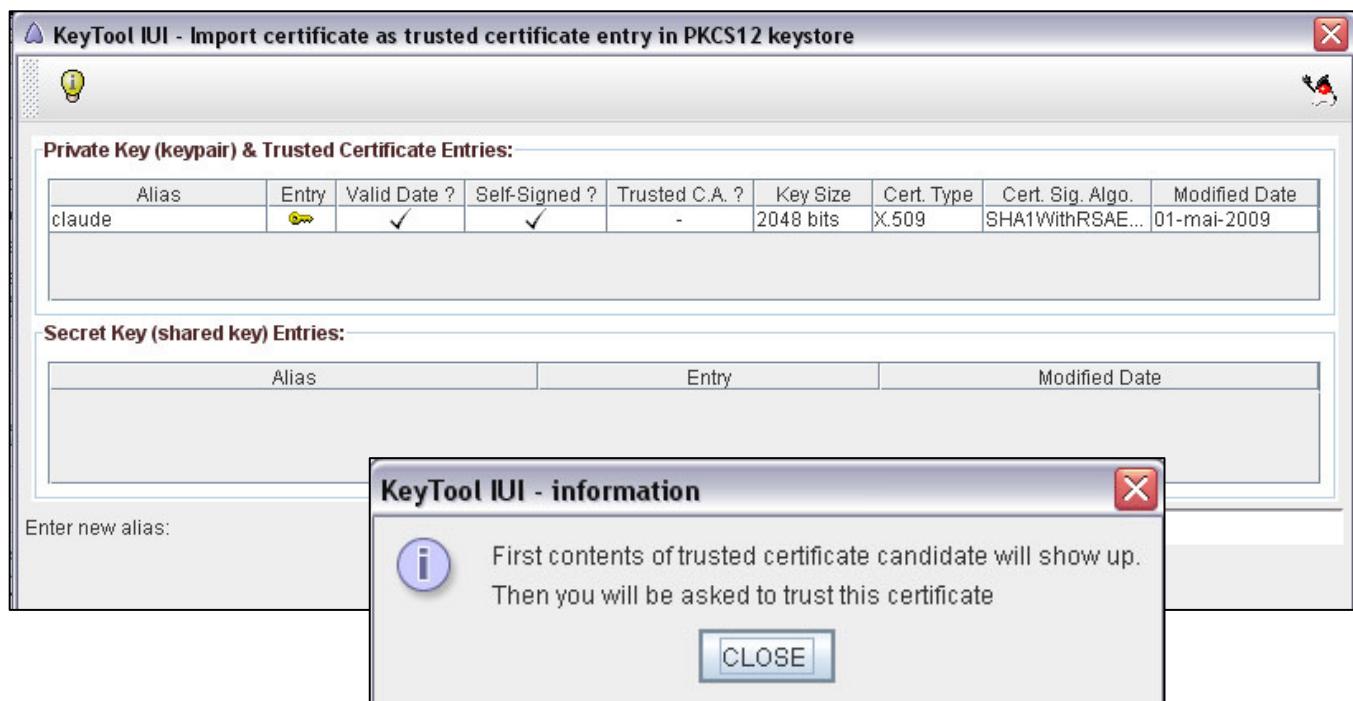
15.6 L'importation d'un certificat

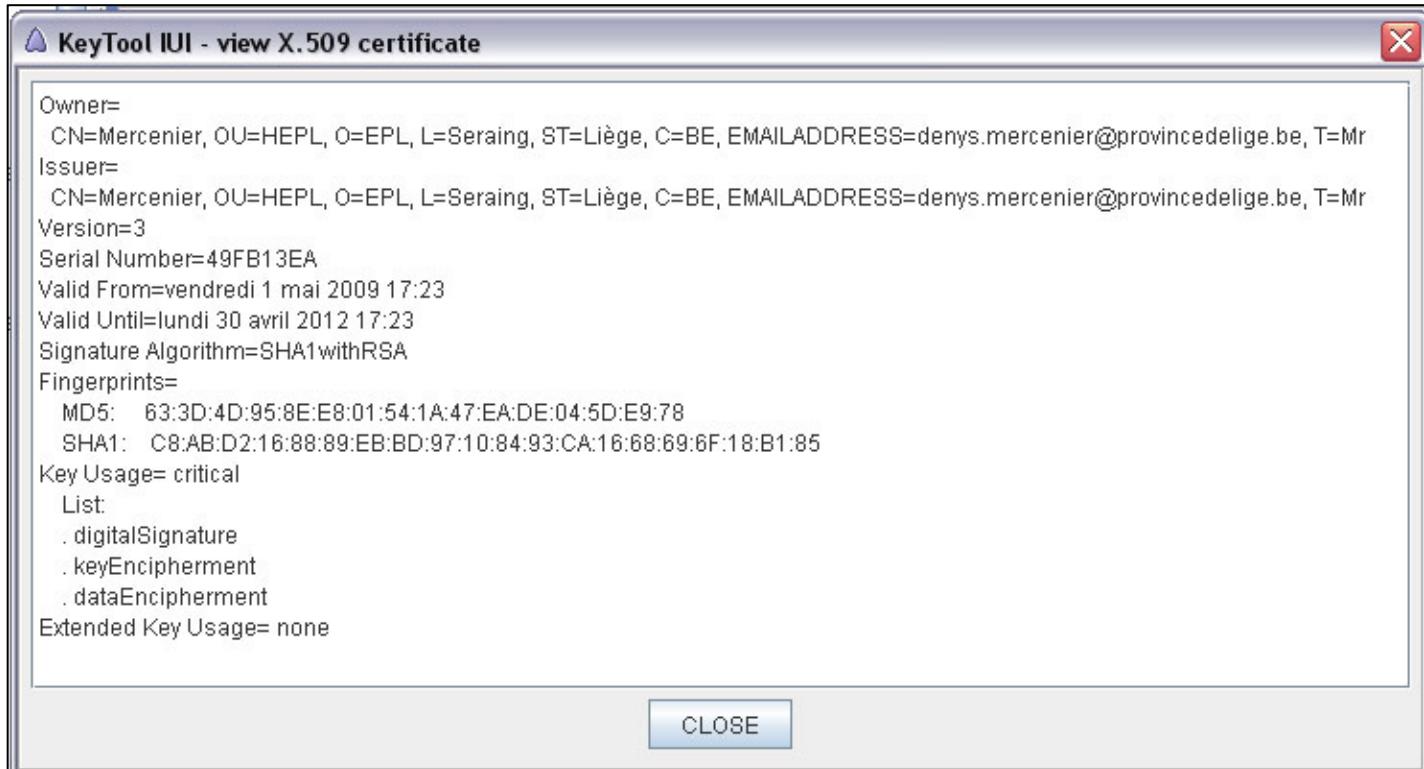
Nous allons enfin importer dans le keystore PKCS12 le certificat que nous venons d'exporter du keystore JKS. Bien normalement, on nous demande :

- ◆ le nom du fichier certificat à importer;
- ◆ le keystore dans lequel on veut l'importer;
- ◆ le mot de passe de ce dernier :

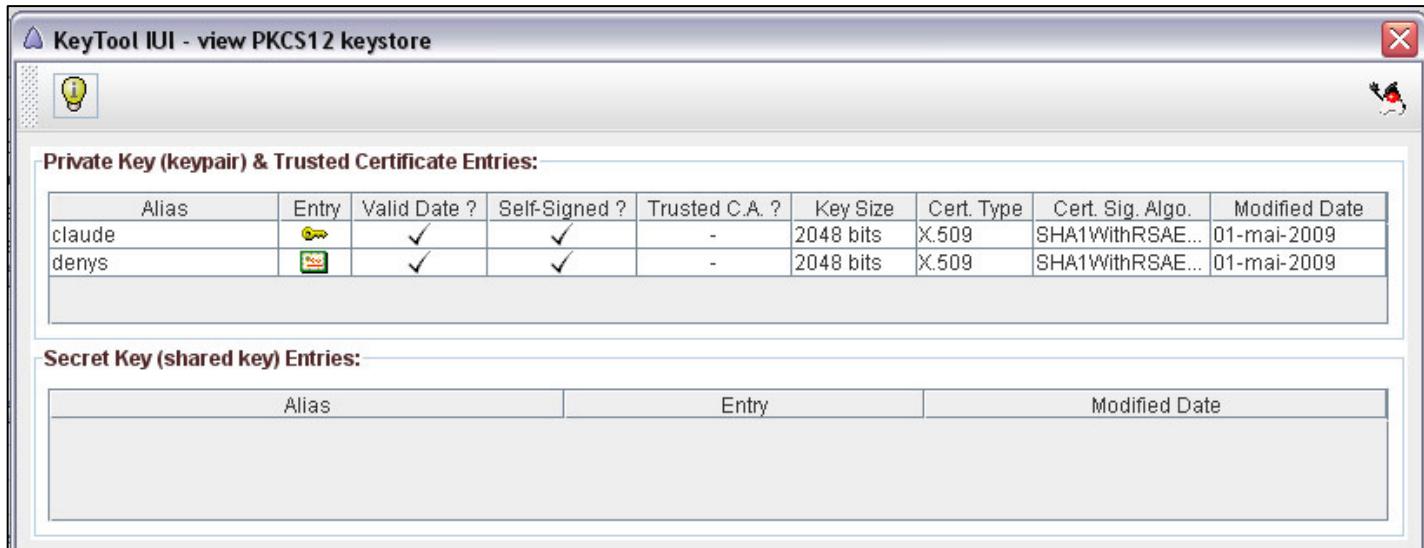


Il faut évidemment préciser l'alias de la nouvelle entrée :





Ultime vérification :



16. L'utilisation des keystores en programmation

16.1 Afficher le contenu d'un keystore

L'outil graphique Keytool IUI est écrit en Java. Il nous est donc bien sûr possible d'obtenir les mêmes résultats que lui dans nos propres applications. Ainsi, pour afficher le contenu d'un keystore, il suffit de savoir que la classe KeyStore possède les méthodes :

- ◆ **public final void load(InputStream stream, char[] password)**
throws IOException, NoSuchAlgorithmException, CertificateException
- qui permet évidemment de créer un objet Keystore à partir d'un fichier keystore protégé par un mot de passe; un premier paramètre à null signifie que l'on veut créer un keystore vide.
- ◆ **public final Enumeration<String> aliases()** throws KeyStoreException
qui fournit bien sûr la liste des alias du keystore
- ◆ **public final boolean isKeyEntry(String alias)** throws KeyStoreException
- ◆ **public final boolean isCertificateEntry(String alias)** throws KeyStoreException
qui permettent évidemment de voir à quel type d'entrée on a affaire
- ◆ **public final Certificate getCertificate(String alias)** throws KeyStoreException
qui fournit le certificat qui se trouve dans la TrustedCertificateEntry si c'en est une, le permier certificat de la chaîne si il s'agit d'une KeyEntry.

Il devient alors aisément de parcourir le keystore PKCS12 utilisé ci-dessus :

AffichageKeystore.java

```
package cryptographienewcryptix;

import java.io.FileInputStream;
import java.security.*;
import java.security.cert.X509Certificate;
import java.util.*;
```

```
/*
 * @author Vilvens
 */
public class AffichageKeystore
{
    public static void main(String[] args)
    {
        try
        {
            KeyStore ks = KeyStore.getInstance("PKCS12", "BC");
            ks.load(new FileInputStream("C:\\java-netbeans-application\\carteid_keystore.p12"),
                    "pwdpwd".toCharArray());

            Enumeration en = ksAliases();
            String aliasCourant = null;
            Vector vectAliases = new Vector();

            while (en.hasMoreElements()) vectAliases.add(en.nextElement());
            Object[] aliases = vectAliases.toArray();
            //OU : String[] aliases = (String []) (vectAliases.toArray(new String[0]));
            for (int i = 0; i < aliases.length; i++)
            {
                if (ks.isKeyEntry(aliasCourant=(String)aliases[i]))
                    System.out.println((i+1) + ".[keyEntry] " + aliases[i].toString());
                else
                    if (ks.isCertificateEntry(aliasCourant))
                        System.out.println((i+1) + ".[trustedCertificateEntry] " + aliases[i].toString());
                X509Certificate certif = (X509Certificate)ks.getCertificate(aliasCourant);
                System.out.println("Type de certificat : " + certif.getType());
                System.out.println("Nom du propriétaire du certificat : " +
                        certif.getSubjectDN().getName());
                System.out.println("Recuperation de la cle publique");
                PublicKey clePublique = certif.getPublicKey();
                System.out.println("**** Cle publique recuperée = "+clePublique.toString());
                System.out.println("Dates limites de validité : [" + certif.getNotBefore() + " - " +
                        certif.getNotAfter() + "]");
            }
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

Le résultat est sans surprise :

1.[keyEntry] claudé

Type de certificat : X.509

Nom du propriétaire du certificat :

T=Mr,E=**claude.vilvens@provincedelige.be**,C=BE,ST=Liège,L=Seraing,O=EPL,OU=HEPL,CN=Vilvens

Recuperation de la cle publique

*** Cle publique recuperée = RSA Public Key
modulus:

996e930fba1369e6d0a08b1f3687bc21614a452010a664d9c421215e9497bce4b84f43a9f2e618
e36f01b138682c90c5694b326b7e694ec0b3ee3597f65f55557216bafbc6f08a2210229bd6c30b
9c3b62748cec9b50390de8ca14e56b720d5d7dd1d70205cebdd191af1ed1e2bce8b723be41730
b04d3de04512e5604d3377599caa69f8713e413feb4351638fcf74a48f5576d0b438715dcbb38a
4bb57e007dcfe7886a2f5c6fd45c8d7893d0fd26264c84496bed9fdd380edeab21b56b98ff9c791
a6721b6ea95e9c1d585c576242ba71a14cc8bcd9a44497895ffd19335ff8f1f1ecb0ad6a4a259fe
86922a522408693f2c2b60de2671ae476c09d359819

public exponent: 10001

Dates limites de validité : [Fri May 01 17:02:39 CEST 2009 - Mon Apr 30 17:02:39 CEST 2012]

2.[trustedCertificateEntry] denys

Type de certificat : X.509

Nom du propriétaire du certificat :

T=Mr,E=**denys.mercenier@provincedelige.be**,C=BE,ST=Liège,L=Seraing,O=EPL,OU=HEPL,CN=Mercenier

Recuperation de la cle publique

*** Cle publique recuperée = RSA Public Key
modulus:

ad429769d497b42fc03f2a8f1663794d40cbfb5dd11b6d8255e0d57d72b3d4040488c865fc561a
7741da64063da462a82b5f01c97373e8ffa6fb2bbe78ef4f89dbf5d447c412270db6a595107cd6e
...

0de4e91d6ca6772c13491c2577cbd5ea29792f8673

public exponent: 10001

Dates limites de validité : [Fri May 01 17:23:22 CEST 2009 - Mon Apr 30 17:23:22 CEST 2012]

16.2 Utilisation des deux keystores dans un processus de signature

Munis des deux keystores créés ci-dessus, nous allons signer un message avec le keystore JKS et vérifier la signature avec le keystore PKCS12 :

SignatureAvecKeystore.java

```
package cryptographienewcryptix;

import java.io.*;
import java.security.*;
import java.security.cert.*;

public class SignatureAvecKeystore
{
    private static String codeProvider = "CryptixCrypto"; // "BC";
```

```

public static void main(String[] args)
    throws KeyStoreException, NoSuchAlgorithmException, SignatureException,
    UnrecoverableKeyException, NoSuchProviderException, InvalidKeyException
{
    try
    {
        KeyStore ks = null;
        ks = KeyStore.getInstance("JKS");
        ks.load(new FileInputStream("C:\\java-netbeans-application\\java_keystore.jks"),
            wdpwd".toCharArray());
        System.out.println("Recuperation de la cle privee");
        PrivateKey cléPrivée;
        cléPrivée = (PrivateKey) ks.getKey("denys", "pwddenys".toCharArray());
        System.out.println(" *** Cle privee recuperée = " + cléPrivée.toString());

        String Message = "Code du jour : CVCCDMMM - bye";
        System.out.println("Message a envoyer au serveur : " + Message);
        byte[] message = Message.getBytes();

        System.out.println("Instanciation de la signature");
        Signature s = Signature.getInstance("SHA1withRSA", codeProvider);
        System.out.println("Initialisation de la signature");
        s.initSign(cléPrivée);
        System.out.println("Hachage du message");
        s.update(message);
        System.out.println("Generation des bytes");
        byte[] signature = s.sign();
        System.out.println("Termine : signature construite");
        System.out.println("Signature = " + new String(signature));

        System.out.println("\nVérification de la signature");
        KeyStore ksv = null;
        ksv = KeyStore.getInstance("PKCS12", "BC");
        ksv.load(new FileInputStream("C:\\java-netbeans-application\\carteid_keystore.p12"),
            "wdpwd".toCharArray());
        System.out.println("Recuperation du certificat");
        X509Certificate certif = (X509Certificate)ksv.getCertificate("denys");
        System.out.println("Recuperation de la cle publique");
        PublicKey cléPublique = certif.getPublicKey();
        System.out.println(" *** Cle publique recuperée = "+cléPublique.toString());
        System.out.println("Debut de verification de la signature construite");
        Signature sv = Signature.getInstance("SHA1withRSA", codeProvider);
        sv.initVerify(cléPublique);
        System.out.println("Hachage du message");
        sv.update(message);
        System.out.println("Verification de la signature construite");
        boolean ok = sv.verify(signature);
        if (ok) System.out.println("Signature testee avec succes");
        else System.out.println("Signature testee sans succes");
    }
}

```

```
        catch (KeyStoreException ex) { ex.printStackTrace(); }
        catch (IOException ex) { ex.printStackTrace(); }
        catch (NoSuchAlgorithmException ex) { ex.printStackTrace(); }
        catch (CertificateException ex) { ex.printStackTrace(); }
    }
}
```

Résultat :

Recuperation de la cle privee

*** Cle privee recuperée = Sun RSA private CRT key, 2048 bits

modulus:

21872064231230364993003540556032474627634761150140615706853706567521117605
185100283266433057527707268978660350493542705153882688165482054909847030780
117261981014439777364184482300096861464547086997401406688612405226692127897
316372297168530245388751229321383517255476838125664840377233223133603117435
200677378841430228437751031982649946877201689809143819358633276089502762329
857743284516110801781758906206923151948912159880868097171974985864480751967
320673841872535405029706476336244268320374054551437471412123669373546272920
265631448813901012024270080350221515064336446710845702144884175829340209564
19965457986979443

public exponent: 65537

private exponent:

106395075711677988918771514247877579249736207862197358550818036433247332487
805377069907592442040119439935299021525766230378347805037086957146740365614
387877259483045304215484402638617696011257169743460287245875817913956162738
093686754562045016143451625658264896624877574491450556345670310717598721086
320667634197183084391272652240791296422973877278660969545365357456500669259
040984464291576101899678469603589508937150709615745465510706372769453027220
545040081153593784582701975711129178815094583375921001049515011641449302527
279438665314444345007075433309726862559932061521746396507925417260360833857
4695171551105665

prime p:

168287685698178599556995264038707092046108186237055170109325215416529276297
247522104502459836867322490629769331026139025929581825533457040576347778629
049620927680115379973315507296711896180636829631403510366458922704841321304
111997802624712997918826755876175131653310617591745152604002802977740173702
427436177

prime q:

129968298871597646019234789640002916974920361441924188140695687035936519992
194437084238453732980438690163572179641724726782674758522904324943618729496
045527606987859849272039556323265114949827699674858839889761253709737586345
760883079510183575132445177965295315038003897453546564630774793987095790043
531685059

prime exponent p:

331686228567583650529885076458791142096766626733607219174230405348933985677
029196182897946764852709860302536040536588155993165454692107449703935831141
406221450912316761999376902780357288701845664029302400690228406026890969572
182839558799367959186030060218281913356701290482105091198239804394993640800
50279889

prime exponent q:

213305311909746293022703113869702820541410715728418537260070312916144042149
632019359761479523313181645696229971501043862440216931301762198314473206655
700702647475688624558655029649364416497603908891585162862851831011785324127
592666494226396468273582912582925127568971713842615140938494847815005617850
71437577

crt coefficient:

167514887051424254013815759052748948648549010143972319046617152481802660375
444457355394613322865978931477185139061062648976272631174679474072107689192
097089719429861531016433500313332806870266940471635065716124932610773686795
275524795135184330725722436770878590374966058476164894533666631893958727227
774338482

Message a envoyer au serveur : Code du jour : CVCCDMMM - bye

Instanciation de la signature

Initialisation de la signature

Hachage du message

Generation des bytes

Termine : signature construite

Signature =

→À ^%5äQÛ^2.¢J¢§Z6^-iLâö5XÿÔ}¤@üu©¾J¢p§?g...!6ê£?pžé2^Í½Ó?£VW?ÝAEžÍK↑ dËð
◀ Æð6Hvg.g!~T^ _Š,I³”“...ðO”“ A
ð·^] | YØ8Dh.Y?-T<Q4{ðx+K^ Öš.. g¹/⁴â-²°¢75?¶|X+D+Ý
8} ,.-ÍÑó;óÐSI¶ð5&) 1...¾€ž:¾GÃPß_ @Mèã} _>~%→ bš | ▲ r; \$yg†÷-1šúž#QÛFt[:du¶
}E.>%o>ÝñG^ÝIèë2,■ + €| 'Y+{Z‡È G| tTv\rÄÄ

Vérification de la signature

Recuperation du certificat

Recuperation de la cle publique

*** Cle publique recuperée = RSA Public Key

modulus:

ad429769d497b42fc03f2a8f1663794d40cbfb5dd11b6d8255e0d57d72b3d4040488c865fc561a
7741da64063da462a82b5f01c97373e8ffa6fb2bbe78ef4f89dbf5d447c412270db6a595107cd6e
7151feb32008c6294f33436d659a0109e9b5268081be9a6247893b58362adebe2dbb07631b060
4a70f4126a809166b893a19a89c6da5674bbbd2c72713bf5045465639c255d5f8905ba753a4ce
dfb6db294cc8dd698cf23aed6d8e54f92ecfc29e8efed05511d841e4e60a9364ad81057d32e2edc
fae2faa84b0d1125aba73f9ebfee25d7c4fcf7210d9b8c3d5ccd7e3ca931676061c2dd83dafe4065
0de4e91d6ca6772c13491c2577cbd5ea29792f8673

public exponent: 10001

Début de vérification de la signature construite

Hachage du message

Vérification de la signature construite

Signature testee avec succes

On peut encore remarquer que la signature ainsi réalisée est une signature CMS/PKCS#7.

17. Un protocole sécuritaire : SSL

17.1 Une sécurisation au niveau des protocoles

Les chiffrements symétriques et asymétriques, les digests et les MACs, les signatures digitales et les certificats constituent les techniques fondamentales que l'on peut mettre en œuvre pour sécuriser une communication d'un point de vue cryptographique. Les protocoles Internet existant avant que ces techniques ne soient stabilisées se sont donc très logiquement vus ajoutées des extensions cryptographiques. Un exemple classique est SSL.

Dans le cas de serveurs Web, on peut évidemment penser à utiliser le header Authorization du protocole **HTTP**, avec son champ

Basic nom:password

Le couple nom/mot de passe envoyé par le client y est codé selon le format **Base64** évoqué ci-dessus. Java, dès le le JDK 1.2, propose des classes BASE64Encoder et BASE64Decoder (package sun.misc) qui permettent de réaliser ou d'inverser ce codage. Ceci dit, un tel algorithme de cryptage utilisé est simple et ne suffit certainement pas à assurer une sécurité de bon niveau. On a donc imaginé mieux ...

17.2 Les protocoles SSL et HTTPS

Le protocole **SSL** (Secure Socket Layer) est un protocole visant à trois fonctionnalités principales :

- ◆ l'authentification des serveurs : un utilisateur doit pouvoir avoir l'assurance qu'il s'adresse bien au serveur visé (ceci prend évidemment tout son sens lorsque, par exemple, il s'agit d'envoyer à un serveur son numéro de carte de crédit);
- ◆ l' authentification des clients : réciproquement, un serveur doit pouvoir s'assurer qu'un client est bien celui qu'il prétend être;
- ◆ le cryptage et le décryptage des données transmises entre un client et le serveur qu'il accède; encore une fois, l'exemple du numéro de carte de crédit est évocateur.

SSL a été conçu par Netscape, en collaboration avec Mastercard, Bank of America, MCI et Silicon Graphics. Sa deuxième version fut intégrée au célèbre browser Navigator de Netscape. La version 3.0 apparut en 1995, pour voir son développement confié à l'IETF en 1999. Le protocole prit alors le nouveau nom de **TLS** (Transport Layer Security Protocol).

Ce qui rend ce protocole intéressant est que SSL se situe entre ce protocole applicatif et la couche transport (soit TCP); *il est donc indépendant des applications qui l'utilisent*. On désigne par le nom d'un protocole applicatif suffixé d'un "s" la version de ce protocole qui utilise SSL. Par exemple, **HTTPS** désigne une "version sécurisée" de HTTP. Donc, en pratique, il suffit de remplacer "http" par "https" dans les URLs concernés. On rencontre de même FTPS, SMTPS, WTLS (WAP Security = Wireless Transport Layer Security), etc.

SSL a donc été au départ développé par Netscape. Mais, à l'heure actuelle, la plupart des navigateurs sont capables de l'utiliser, à commencer par Internet Explorer. Java, de son côté, fournit une extension prenant SSL en compte : elle se nomme JSSE et comporte des classes comme SecureSocket. A vrai dire, de telles classes ne sont nécessaires que si l'on désire travailler en dessous de la couche application. En effet, au niveau de celle-ci, il suffit d'utiliser "https" dans les constructeurs d'URL !

17.3 Le dialogue SSL de création d'une session

Qui dit authentification dit bien sûr certificat. En pratique, SSL en distingue deux types :

- ◆ du côté du browser sont installés des "**root certificates**" : un tel certificat a pour rôle de spécifier au browser qu'il peut accepter tous les certificats authentifiés par le propriétaire du "root certificate"; bon nombre de browsers possèdent un certificat root pour Verisign, ce qui donne déjà un large spectre puisque bon nombre de certificats sont créés sous son autorité.
- ◆ du côté du serveur se trouvent les "**server certificates**" (of course !) : ce sont les certificats obtenus par CSR et validés par un CA; le browser d'un client ne pourra alors accéder à un serveur particulièrement méfiant (c'est-à-dire ayant demandé l'authentification d'un client) que s'il possède le certificat root du CA en question.

Le dialogue de début de session SSL en fait une combinaison de chiffrement asymétrique et symétrique qui sous-tend le mécanisme :

- ◆ le client contacte le serveur Web sécurisé par une URL https;
- ◆ le serveur envoie au client un **certificat** contenant sa clé **publique**;
- ◆ le client peut vérifier l'authenticité de ce certificat au moyen de ses certificats root; en cas de succès, le client construit un message (pre-master secret) qui est ensuite **crypté** au moyen de la clé **publique** du serveur et lui envoie ce message; il génère à partir du message un *master secret* qui fournit une **clé de session**;
- ◆ le serveur déchiffre le message au moyen de sa clé **privée**; à partir de ce message, il génère, de la même manière, un *master secret* qui donne une **clé de session (master secret)** qui est forcément la même que celle générée sur le client;
- ◆ le client et le serveur communiquent par **chiffrement symétrique** en utilisant la clé de session.

Une variante : le client envoie également au serveur un certificat contenant sa clé publique. Le point intéressant est que la clé de session, négociée par les deux intervenants, **n'a de sens qu'au sein de la session ainsi créée** : un hacker équipé du même certificat pourrait certes obtenir la clé publique, mais il se verrait attribuer une autre clé de session, ne lui permettant pas d'intervenir dans une autre communication.

17.4 Les sous-protocoles de SSL

Ce qui vient d'être décrit désigne en fait le sous-protocole de SSL connu sous le nom de "**SSL Handshake**". En pratique,

- ◆ il est précédé d'un sous-protocole "SSL Change Cipher SPEC" qui permet notamment au serveur et au client de s'accorder sur les algorithmes de chiffrements asymétrique (en pratique, par exemple, RSA) et symétrique (par exemple, DES) ainsi que sur l'algorithme d'authentification (par exemple, SHA-1 ou MD5);
- ◆ le sous protocole "SSL Alert" gère les messages d'erreur entre le client et le serveur;
- ◆ le sous protocole "SSL Record" gère l'échange des données cryptées et authentifiées; en fait, il fractionne les données en blocs de taille inférieure ou égale à 16 Ko et chaque bloc est accompagné d'un MAC qui permettra d'en tester l'authenticité.

SSL sera étudié plus en profondeur dans le volume IV consacré aux protocoles et technologies de l'e-commerce.

18. Les jars signée

C'est l'historique et la préoccupation même de Java : le bytecode reçu par le web (par exemple une applet au sein d'une page HTML) pourrait se montrer dangereux pour le système et est donc cantonné à la sandbox. On pourrait évidemment imaginer d'assouplir ses conditions d'actions si on était sûr que ce code

- ◆ n'a pas été modifié par un hacker;
- ◆ a bien été construit par une entité connue.

On s'en doute, ces conditions peuvent être remplies si le créateur du code Java possède un certificat fourni par un CA. En effet, dans ce cas, *l'auteur va signer son code au moyen de sa clé privée*. Il sera alors possible pour celui qui reçoit ce code de vérifier les points cruciaux énoncés ci-dessus.

On pourra atteindre un tel but en construisant un fichier jar contenant, outre les fichiers class et les images (les sons, les séquences vidéos) utilisées par ce code, des éléments d'authentification du type évoqué ci-dessus. L'utilitaire **jarsigner**, apparu avec le JDK 1.2, permet précisément d'orner un fichier jar d'une signature électronique et, par la suite, de vérifier de telles signatures ainsi que l'intégrité même du fichier jar. La signature qu'il utilise est générée à partir d'une clé privée figurant dans un keystore. Dans sa version actuelle, jarsigner peut utiliser DSA avec SHA-1 comme algorithme de digest ou RSA avec MD5 (mais dans ce dernier cas, un provider pour RSA sera nécessaire).

Le fichier jar "signé" contiendra une copie du certificat associé à cette clé secrète au sein du keystore. Très logiquement, l'outil jarsigner doit donc connaître :

- ◆ l'emplacement du keystore utilisé;
- ◆ l'alias dans le keystore qui correspond à la clé privée qui sera utilisée;
- ◆ le mot de passe du keystore;
- ◆ le mot de passe de la clé privée.

En pratique, considérons donc une applet quelconque, par exemple `forcerAvenir.java` (peu importe son effet ici), avec un bytecode `forcerAvenir.class`. Après avoir ajusté le path :

```
| C:\java-application\ForcerAvenirSigne>set path=c:\jdk12\bin;%path%
```

confectionnons le fichier jar correspondant :

```
| C:\java-application\ForcerAvenirSigne>jar cvf forcer.jar forcerAvenir.class  
added manifest  
adding: forcerAvenir.class(in = 1062) (out= 621)(deflated 41%)
```

Nous allons à présent construire un nouveau jar, mais signé au moyen de la clé privée contenue dans le keystore (`c:\windows\keystore`) pour l'alias claude :

```
| C:\java-application\ForcerAvenirSigne>jarsigner -keystore c:\windows\keystore forcer.jar  
claude  
Enter Passphrase for keystore: beaugosse
```

On peut constater que le nouveau jar a bien été créé en vérifiant le contenu :

```
C:\java-application\ForcerAvenirSigne>jar tf forcer.jar  
META-INF/MANIFEST.MF  
META-INF/CLAUDE.SF  
META-INF/CLAUDE.DSA  
META-INF/  
forcerAvenir.class
```

Le fichier **MANIFEST** contient à présent, pour chaque fichier, trois informations précisant le nom du fichier, le nom de l'algorithme de digest utilisé (par exemple, SHA) et la valeur de ce digest, calculée d'après le contenu binaire du fichier source :

```
Created-By: 1.2.2 (Sun Microsystems Inc.)
```

```
Name: forcerAvenir.class
```

```
SHA1-Digest: +7VjhgeuydLSsoQxyEhWLYi4s4U=
```

Les deux fichiers complémentaires apparus dans le répertoire META-INF sont le fichier de signature (Signature File – **SF**) comportant les informations de digest pour le fichier class et le fichier de signature de bloc (Digital Signature Algorithm – **DSA**) qui contient la signature associée au fichier .SF et, sous forme codée, le certificat (ou la chaîne de certificats) qui permettra d'obtenir la clé publique associée à la clé privée utilisée pour cette signature (c'est évidemment indispensable pour la vérification).

En pratique, la vérification manuelle d'un fichier jar est simple à commander :

```
C:\java-application\ForcerAvenirSigne>jarsigner -verify forcer.jar  
jar verified.
```

Dans le cas d'une application signée, cette vérification sera prise en charge par l'environnement Java. et finalement le browser si le code tourne dans une page HTML. Tout ceci réclamerait un assez long exposé, qui sort du cadre de ce chapitre introductif (voir Volume IV) ...

Signatures et certificats : les maîtres-mots ! Et pourtant, certains s'en passent ...

19. Le protocole Kerberos

Développé à l'origine au MIT dans les années '80 pour les systèmes distribués Unix (projet Athena), puis retrouvant une seconde jeunesse avec Windows 2000 puis 2003, le protocole Kerberos est un protocole d'authentification de clients vis-à-vis de serveurs qui n'utilise, en plus des digests, que des clés symétriques et non pas asymétriques, sans envoyer sur le réseau des données qui pourraient être utilisées pour une attaque. Il n'est donc pas fait usage de certificats ! La version 5 utilisée actuellement date de 1989.

19.1 Le problème de l'authentification sans PKI

Kerberos a été créé au MIT et est basé sur le principe des secrets partagés : si un secret n'est connu que par deux intervenants, chacun peut vérifier l'identité de l'autre en vérifiant qu'il connaît le secret en question.

Pour Kerberos, le secret est en fait une **clé symétrique**. Elle va permettre de créer des **authentificateurs** : il s'agit d'une information (classiquement, des informations sur le nom de l'intervenant à authentifier et la date-heure courante) cryptée au moyen de la clé symétrique. Un authentificateur doit varier en fonction du temps, de manière à éviter qu'il puisse être

réutilisé par un hacker qui l'aurait capté. Mais un autre problème est évident : comme les deux intervenants peuvent-ils se procurer une clé commune en toute généralité ? De plus, un client doit, en principe, posséder une clé commune pour chaque serveur – et un serveur doit posséder les clés de tous ses clients : le risque de toutes ces clés mémorisées sur les machines est évident et amène à l'idée de centraliser tout cela ...

Le nom de Kerberos vient de Cerbère, le mythique chien à trois têtes qui gardait l'entrée des enfers. Ici, les trois têtes vont être le **client C**, le **serveur S** auquel il veut se connecter et un intermédiaire : **le serveur de clés** ou centre de distribution des clés ou **KDC (Key Distribution Center)**. L'idée de base est qu'un client peut prouver qu'il est un utilisateur connu en prouvant qu'il connaît la clé secrète connue seulement de cet utilisateur et du serveur KDC – ce dernier est donc en fait un "**serveur d'authentification**".

Le KDC est donc une application serveur qui attribue et mémorise dans une BDD les clés correspondant à des clients qui se connectent au réseau (comme d'ailleurs aussi aux serveurs qui font partie de l'infrastructure). Ces clés sont des **clés symétriques à long terme** (long-term key) : elles ne changent pas au cours d'un long laps de temps et sont, dans le cas d'un client, dérivées d'un mot de passe de ce client : la clé est en fait le résultat d'un hashage du mot de passe de type MD5. Ce n'est pas cette clé qui va servir au client et au serveur pour s'authentifier : mais elle va leur permettre de recevoir chacun du KDC une **clé de session, à usage limité dans le temps**, qui, elle, va leur permettre de s'authentifier.

On s'attendrait à ce que le KDC envoie au client et au serveur la clé de session telle quelle. Mais, dans ce cas, le serveur devrait conserver en mémoire les clés de tous les clients qui, potentiellement, pourraient l'appeler. De plus, une requête client pourrait précéder la réception par le serveur de la clé de session correspondante. Tout ceci apporte de gros problèmes d'efficacité et Kerberos a imaginé un système plus efficace.

19.2 La version à 3 acteurs

Ces trois acteurs sont : le client C, le serveur KDC qui possède les clés secrètes de tout le monde et le serveur S.

1) Interaction client et serveur KDC

1.1) Le client envoie au serveur KDC une requête comportant son identifiant (le nom nc avec un password pwd), sa date-heure (t), son adresse réseau (a) et le nom du serveur auquel il souhaite accéder (ns).

1.2) Le serveur KDC vérifie l'identité du client. En cas de succès, il crée une clé de session **Kcs**. Il envoie au client :

- la clé de session Kcs, avec la version (v) et nom du serveur (ns), le tout crypté avec la clé secrète **Kc** du client (celui-ci pourra ainsi ordonner ses tickets sur base de ces informations);
- un "ticket" **Tcs**, qui est un message crypté avec la clé secrète **Ks** du serveur (donc, incompréhensible pour le client) et qui contient principalement la clé de session Kcs, le nom du client, les limites de validité du ticket dans le temps tv , des flags, etc.

Le ticket est donc en fait une espèce de certificat, puisque ses composantes stratégiques sont cryptées par le KDC, grand maître de l'authentification.

1.3) Le client peut décrypter la clé de session **Kcs** en utilisant sa clé secrète. Cette clé de session et le ticket sont conservés en cache.

L'idée de base est que le client va pouvoir être authentifié sur base de cette clé de session dont le client dispose déjà et dont le serveur disposera chaque fois qu'il décryptera le ticket avec sa propre clé.

Donc ...

2) Interaction client et serveur

2.1) Le client envoie ensuite au serveur S une requête comportant

- un **authentificateur** – donc en gros son nom, sa date-heure courante t et une somme de contrôle (checksum ck), le tout crypté au moyen de la clé de session;
- le ticket qu'il avait obtenu du serveur KDC;

2.2) Le serveur décrypte le ticket avec sa clé symétrique **Ks** et en extrait la clé de session; il utilise cette clé de session pour décrypter l'authentificateur; si les informations d'identification concordent (donc notamment la somme de contrôle) et si le temps de validité n'est pas dépassé, l'accès est donné au client.

Eventuellement, si le client veut être sûr de l'identité du serveur, il attend une réponse du serveur : celle-ci comporte la date-heure du client que le serveur a décrypté et qu'il a recrypté avec la clé de session.

19.3 Représentation schématique et notations

Nous pouvons donner un schéma récapitulatif de tout cela. Mais le besoin d'une notation ad hoc se fait alors cruellement sentir. Nous utiliserons ici des éléments d'une notation assez standard, déjà entrevue ci-dessus :

K : clé

Ka : clé symétrique de a

Kab : clé symétrique partagée entre a et b

{X}Ka : message clair X crypté avec la clé symétrique Ka

Tab : ticket de a pour accéder à b

Aab : authentificateur de a vérifiable par b

PKa : clé publique d'un cryptage asymétrique appartenant à a

{X}PKa : message clair X crypté avec la clé publique asymétrique PKa

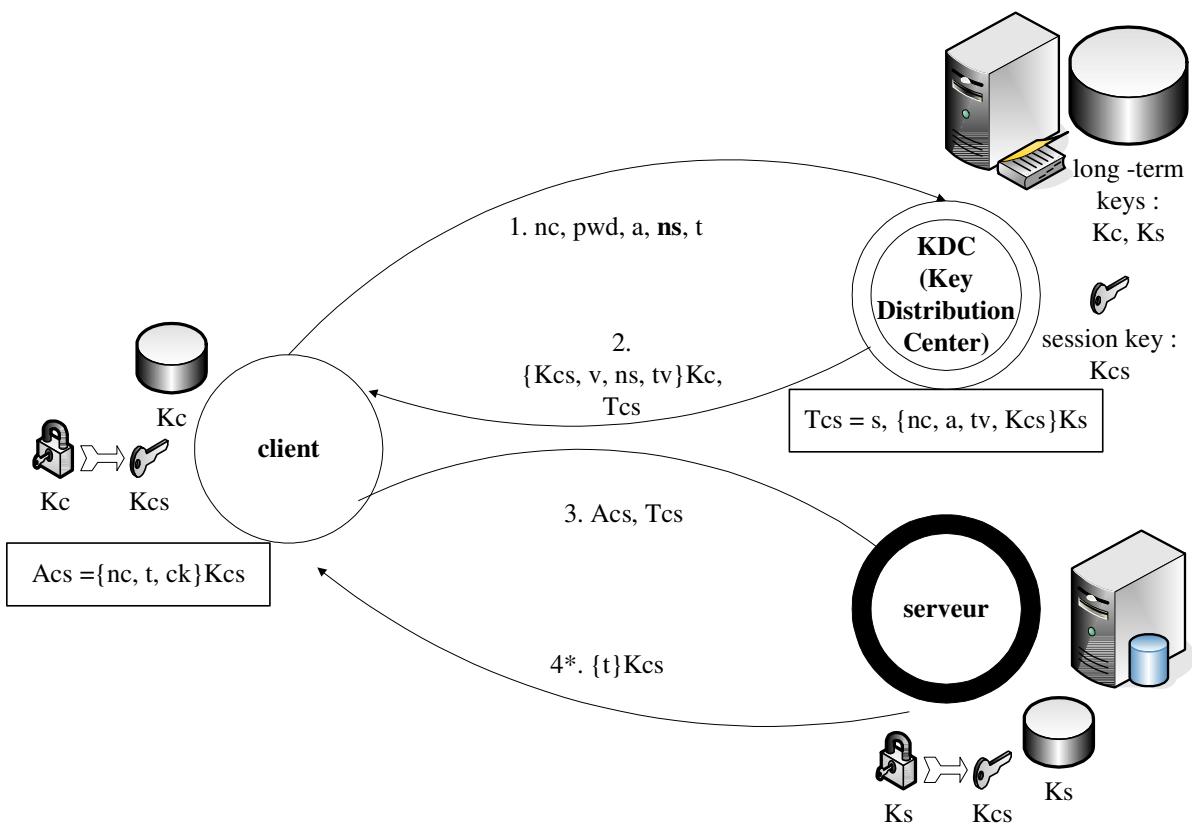
PrKa : clé privée d'un cryptage asymétrique appartenant à a

{X}PrKa : message clair X crypté avec la clé privée asymétrique PrKa

H(X) : hashage du message X

S(X, PrKa) : signature par a du message X en utilisant la clé privée PrKa

Cela donne alors :



19.4 Réflexions et évolution

Ce qui est remarquable dans ce scénario, c'est que :

- ♦ le serveur KDC ne stocke pas les clés de session : le client a l'entièvre responsabilité de fournir le ticket qui a été validé par le serveur KDC et qui contient la clé de session;
- ♦ dans les limites du temps de validité (typiquement, il est de l'ordre d'une session classique, soit 8 h), le client ne doit plus faire appel au serveur KDC; il lui suffit de réutiliser le même ticket.

Evidemment, avec un tel système, le client devra réintroduire son mot de passe à chaque connexion sur un nouveau serveur. Garder ce mot de passe dans la cache du KDC serait évidemment dangereux. Il serait plus intéressant de décharger le KDC d'un travail répétitif d'authentification

- ♦ en lui permettent de créer une seule fois un **ticket de base**, le "ticket granting-ticket" (TGT), qui aurait un temps de validité limité (typiquement, il est de l'ordre d'une session classique, soit 8 h), et
- ♦ en introduisant dans le scénario un deuxième serveur dont le rôle se bornerait à vérifier ce ticket et à attribuer un ticket pour le serveur visé. Ce deuxième serveur est appelé un "**ticket granting server**" (TGS).

19.5 La version à 4 acteurs

Dans ces conditions, le rôle du serveur KDC se limite à fabriquer le TGT si le client est reconnu sur base du mot de passe, puis à effacer ce mot de passe. Le TGT servira au client à accéder le TGS de manière authentifiée, celui-ci lui fournissant le ticket vers le serveur désiré.

1) Interaction client et serveur KDC

1.1) Le client envoie au serveur KDC une requête comportant son identifiant (le nom nc avec un password pwd), sa date-heure (t), son adresse réseau (a) et le nom du serveur TGS auquel il souhaite accéder (tgs).

1.2) A cette requête du client, le serveur KDC va donc répondre en envoyant :

- la clé de session **Kc,tgs**, avec la version (v) et nom du serveur TGS (tgs), le tout crypté avec la clé secrète Kc du client; cette clé est encore appelée la "clé d'entrée en session" [*logon session key*];
- un **ticket Tc,tgs**, qui est un message crypté avec la clé secrète Ktgs du serveur (donc, incompréhensible pour le client) et qui contient principalement la clé de session Kc,tgs, le nom du client, les limites de validité du ticket dans le temps tv, des flags, etc.

Une fois ce ticket Tc,tgs spécial reçu, le client peut oublier sa clé à long terme : il utilisera son ticket TGT dans toutes les requêtes suivantes visant à obtenir un ticket pour un autre serveur quelconque. A priori, le rôle de serveur de tickets peut donc être assuré par un serveur distinct du KDC : c'est le serveur **TGS** qui constitue le 4^{ème} acteur.

La suite de l'histoire devient prévisible ...

2) Interaction client et serveur TGS

2.1) Le client envoie ensuite au serveur TGS une requête comportant

- un **authentificateur** – donc son nom, sa date-heure courante t et une somme de contrôle (checksum ck), le tout crypté au moyen de la clé de session Kc,tgs;
- le **ticket Tc,tgs** qu'il a obtenu du serveur KDC;
- le **nom ns** du serveur si il y a plusieurs serveurs.

2.2) Le serveur TGS décrypte le ticket avec sa clé symétrique Ktgs et en extrait la clé de session Kc,tgs; il utilise cette clé de session pour décrypter l'authentificateur; si les informations d'identification concordent (donc notamment la somme de contrôle) et si le temps de validité n'est pas dépassé, il envoie au client

- la clé de session **Kcs**, avec la version (v) et nom du serveur (ns), le tout crypté avec la clé secrète Kc,tgs;
- un **ticket Tcs**, qui est un message crypté avec la clé secrète Ks du serveur (donc, incompréhensible pour le client) et qui contient principalement la clé de session Kcs, le nom du client, les limites de validité du ticket dans le temps tv, des flags, etc.

3) Interaction client et serveur

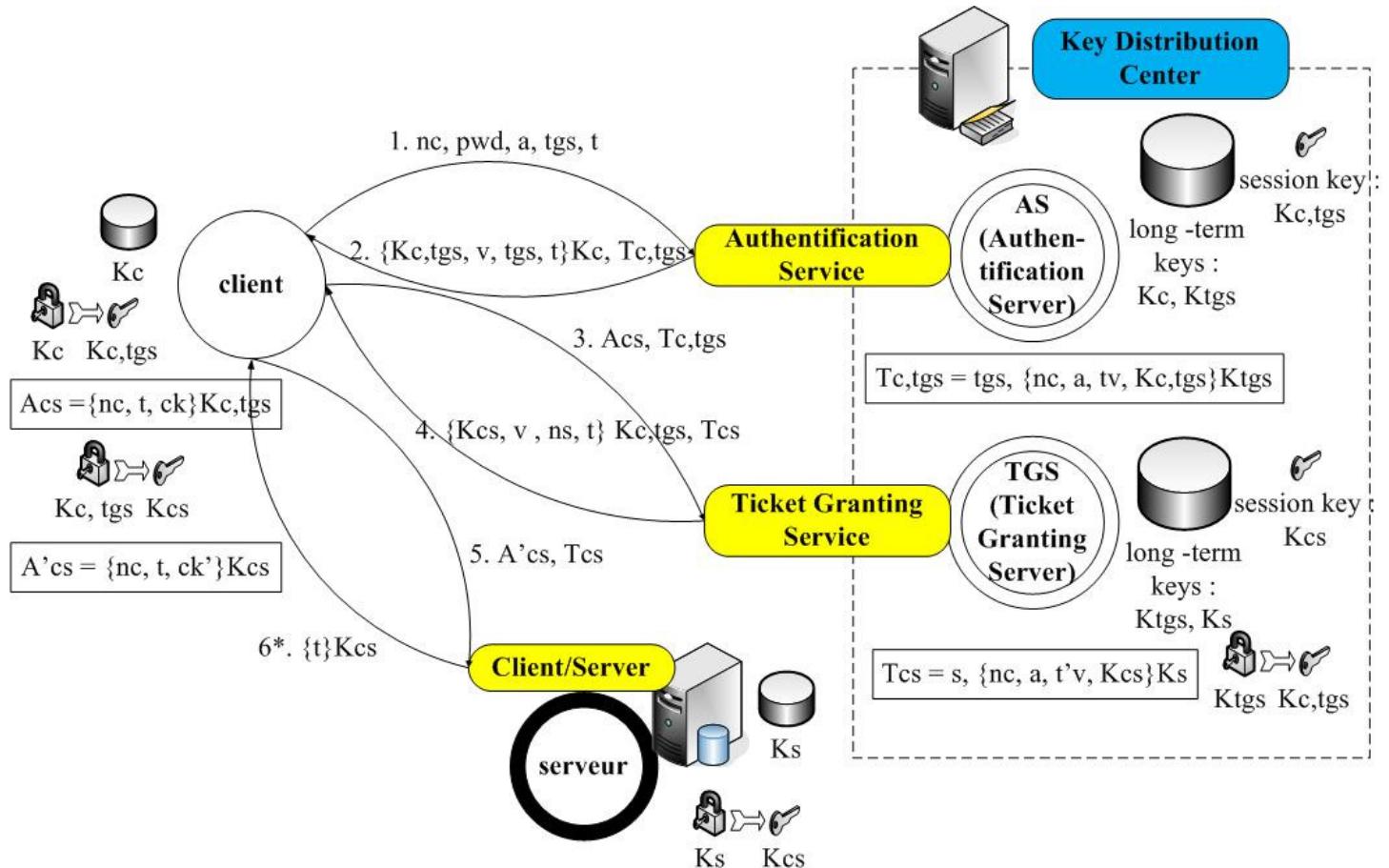
3.1) Le client envoie ensuite au serveur une requête comportant

- un **authentificateur** – donc son nom, sa date-heure courante t et une somme de contrôle (checksum ck'), le tout crypté au moyen de la clé de session Kcs;
- le **ticket Tcs** qu'il a obtenu du serveur KDC;

3.2) Le serveur décrypte le ticket avec sa clé symétrique K_s et en extrait la clé de session K_{cs} ; il utilise cette clé de session pour décrypter l'authentificateur; si les informations d'identification concordent (donc notamment la somme de contrôle) et si le temps de validité n'est pas dépassé, l'accès est donné au client.

Eventuellement, si le client veut être sûr de l'identité du serveur, il attend une réponse du serveur : celle-ci comporte la date-heure du client que le serveur a décrypté et qu'il a recrypté avec la clé de session K_{cs} .

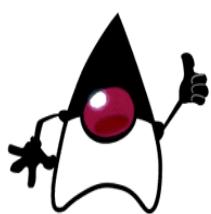
Schématiquement :



19.6 Les sous-protocoles de Kerberos

A la lumière de ce qui vient d'être dit, on peut donc décomposer Kerberos en trois sous-protocoles (qui peuvent alors correspondre à trois services distincts) :

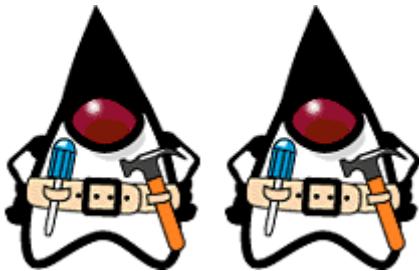
- ◆ **Authentification Service (AS)** : l'attribution au client par le serveur KDC d'un ticket TGT et donc d'une clé de session de type "logon session key";
- ◆ **Ticket Granting Service (TGS)** : l'attribution au client par le serveur TGS d'un ticket et d'une clé de session "session key" pour un serveur donné;
- ◆ **Client/Server (CS)** : le client utilise son ticket pour pouvoir se connecter au serveur visé.



Il reste encore beaucoup à dire sur ce passionnant sujet de la sécurité : ainsi en est-il des applets signées, des chargeurs de classes et des security managers, fers de lance de Java – ce sera pour un autre volume ;-).

Dans un autre registre, Java propose aux développeurs une multitude de classe utilitaires et d'outils destinés à répondre aux problèmes pratiques de l'informaticien en entreprise ☺ ! Nous en connaissons déjà quelques-uns – en voici d'autres ...

XIV. Classes utiles et utilitaires : la suite



Un langage de programmation est sensé être une façon conventionnelle de donner des ordres à un ordinateur. Il n'est pas sensé être obscur, bizarre et plein de pièges subtils (ça, ce sont les attributs de la magie).

(D. Small; ST magazine)

Le chapitre VII nous avait offert une première palette de classes pratiques et d'utilitaires du JDK facilitant la vie de l'informaticien. A sa suite, voici donc un "fourre-tout volume 2" !

1. L'internationalisation

1.1 Un objectif polyglotte

Tout développeur professionnel est bien conscient de ce problème : si l'on désire commercialiser une application, on a tout intérêt à ce qu'elle soit facilement adaptable aux habitudes locales spécifiques des clients, donc notamment à la langue utilisée par ceux-ci. Le problème a déjà été évoqué à propos des dates (chapitre VI), mais c'est évidemment un peu court ...

L'objectif est donc clairement de se doter d'un outil permettant d'adapter facilement une application d'une langue dans une autre. Considérons donc par exemple ceci :

Votre nom s.v.p. :	Dugenou
Date d'enregistrement :	dimanche 15 mai 2005 23 h 19 CEST
Destination :	Djibouti
Date de départ :	31/5/2005

Ok Annuler

On conçoit sans peine qu'un voyageur britannique ou américain préfèrera trouver un interface parlant anglais ... C'est cela l'objectif de l'internationalisation.

Mais on entrevoit aussi avec horreur le travail brut à entreprendre : prévoir dans l'application les différentes phrases possibles alors que, pour l'instant, elles sont écrites dans le code "à la dure" et en français. Ce travail ne peut se faire qu'avec l'aide de traducteurs qui, bien sûr, ne peuvent oeuvrer dans ce code. Mais il y a une technique ad hoc, somme toute assez prévisible ...

1.2 Les fichiers Properties

L'idée est assez simple. On va en fait associer à chaque élément du GUI à adapter un identifiant unique; par exemple, le texte qui affiche, en français, "Votre nom s.v.p. :" se verra associé à l'identifiant "InviteNom" tandis que le texte du bouton Annuler sera identifié par "TexteBoutonAnnuler". Les textes français du GUI peuvent ainsi être mis en correspondance avec les éléments du GUI dans un fichier properties du type suivant (créé par exemple avec Netbeans) :

TextesAeroport_fr_FR.properties

```
# Sample ResourceBundle properties file
InviteNom=Votre nom s.v.p. :
InfoDateEnregistrement=Date d'enregistrement :
InviteDestination=Destination :
InviteDateD\u00E9part=Date de d\u00E9part :
TitreReservation=A\u00E9roport d'Oupeye : r\u00E9servations
TexteBoutonOk=Ok
TexteBoutonAnnuler=Annuler
```

Le nom donné au fichier mérite attention, car il est construit en fonction de ce qu'attendent les classes utilitaires de la librairie Java que nous allons utiliser : les suffixes suivant le nom de base désignent en effet respectivement la langue et le pays considéré selon les codes ISO internationaux⁵. Ainsi, "fr" et "FR" font évidemment référence au français et à la France. Il nous sera alors facile d'instancier un objet **Locale** (classe déjà rencontrée au chapitre VI) en utilisant son constructeur :

```
public Locale (String language, String country)
```

qui demande précisément ces codes ISO (précédemment, nous utilisions des constantes de classe de Locale, comme FRANCE ou FRENCH qui sont en fait construites de cette manière). Par exemple :

```
| Locale localeCourant = new Locale ("fr", "FR");
```

Cet objet sera évidemment chargé de représenter les choix locaux.

Bien sûr, il reste à créer des fichiers properties analogue pour les autres langues que l'on souhaite proposer à l'utilisateur – par exemple :

TextesAeroport_en_UK.properties

```
# Sample ResourceBundle properties file
InviteNom=Please enter your name :
InfoDateEnregistrement=Date of registration :
InviteDestination=Destination :
InviteDateD\u00E9part=Date of departure :
TitreReservation=Oupeye Airport : reservations
TexteBoutonOk=All right
TexteBoutonAnnuler=Cancel
```

1.3 Le principe des bundles

Comment une application à vocation polyglotte peut-elle accéder à ces ressources linguistiques de manière standardisée ? En utilisant un objet **ResourceBundle** du package java.util : un tel objet a pour vocation de permettre l'accès à des ressources contenues dans des fichiers properties du type de ceux décrits ci-dessus. Pour que cela soit possible, il faut créer

⁵ voir <http://www.ics.uci.edu/pub/ietf/http/related/iso639.txt>
et http://www.chemie.fu-berlin.de/diverse/doc/ISO_3166.html

un tel objet dynamiquement en lui transmettant le nom de base du fichier properties (dans notre cas, "TextesAeroport") et un objet Locale qui identifie les choix de l'utilisateur. Comme le choix se fait au moment de l'exécution, ce n'est pas un constructeur qui nous est nécessaire, mais une *factory* :

```
public static final ResourceBundle getBundle (String baseName, Locale locale)
```

Dans notre cas, cela donnerait :

```
| ResourceBundle res = ResourceBundle.getBundle("TextesAeroport", localeCourant);
```

Pour être précis, cette méthode getBundle() essaie tout d'abord de charger la classe dont le nom est le premier argument; ce n'est qu'en cas d'échec (ce qui sera le cas pour nous) que la recherche sera relancée sur un fichier portant ce nom avec l'extension "properties". Une fois un tel objet acquis, on peut obtenir une ressource quelconque en passant l'identifiant correspondant à la méthode :

```
public final String getString (String key)
```

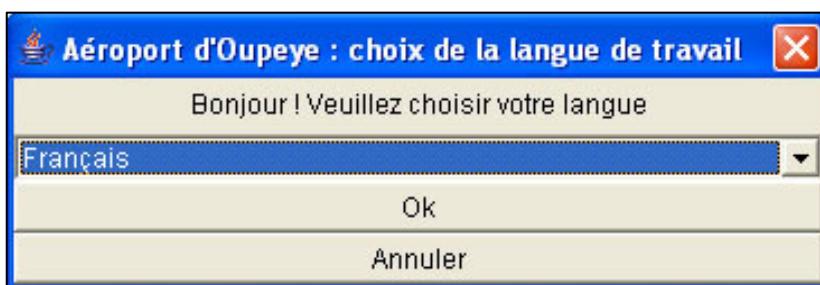
Donc, par exemple, pour que le bouton BAnnuler prenne son texte en français :

```
| BAnnuler.setLabel(res.getString("TexteBoutonAnnuler"));
```

Il nous reste à mettre tout cela en pratique ...

1.4 Une boîte de dialogue de choix de langue

Nous allons tout d'abord considérer que notre application démarre en faisant apparaître une boîte de dialogue demandant à l'utilisateur de choisir sa langue (nous simplifions ici quelque peu en supposant que langue et pays sont les mêmes – l'amélioration à apporter est immédiate) :



Nous serons donc ainsi en possession d'une chaîne de caractère comme "English" ou "Français" – la boîte possèdera getLangueChoisie() pour fournir cette langue à la fenêtre de l'application qui apparaîtra à la fermeture de la boîte. Pour obtenir dans la suite les codes ISO correspondants, notre boîte de dialogue générera deux Hashtables qui établiront les associations entre, par exemple, "Français" et "fr". Sans plus de commentaires, la classe sera donc :

DialChoixLangue.java

```
/*
 * DialChoixLangue.java
 * Created on 15 mai 2005, 18:25
 */
/**
 * @author Vilvens
 */
import java.util.*;

public class DialChoixLangue extends java.awt.Dialog
{
    public static final String FR = "Français";
    public static final String GB = "English";
    public static Hashtable codeLangue, codePays;
    static
    {
        codeLangue = new Hashtable(); codePays = new Hashtable();
        codeLangue.put(FR, "fr"); codePays.put(FR, "FR");
        codeLangue.put(GB, "en"); codePays.put(GB, "UK");
    }
    private static String langue = FR;

    public DialChoixLangue(java.awt.Frame parent, String title, boolean modal)
    {
        super(parent, title, modal);
        initComponents();
        CBLangues.add(DialChoixLangue.FR);
        CBLangues.add(DialChoixLangue.GB);
    }

    public static String getLangueChoisie()
    {
        return langue;
    }

    private void initComponents() { ... }

    private void BOkActionPerformed(java.awt.event.ActionEvent evt)
    {
        setVisible(false);
        langue = CBLangues.getSelectedItem();
        //dispose();
    }

    private void closeDialog(java.awt.event.WindowEvent evt)
    {
        setVisible(false); dispose();
    }
}
```

```

private java.awt.Button BOk;
private java.awt.Button BAnnuler;
private java.awt.Choice CBLangues;
private java.awt.Label label1;
}

```

1.5 Une application internationale

La fenêtre principale de notre application fera donc apparaître tout d'abord la boîte de dialogue décrite ci-dessus. Une fois le choix de la langue posé, le constructeur de la fenêtre principale n'a plus qu'à :

- ◆ récupérer la langue choisir sous forme de String;
- ◆ créer un objet Locale avec les codes ISO associés à cette langue (ces codes sont obtenus au moyen des objets Hashtable de la boîte de dialogue);
- ◆ se procurer un objet ResourceBundle donnant au fichier properties adéquat;
- ◆ utiliser cet objet pour obtenir les textes associés aux divers contrôles.

En pratique :

FenAppInternational.java

```

/*
 * FenAppInternational.java
 * Created on 15 mai 2005, 18:17
 */
/**
 * @author Vilvens
 */

import java.util.*;
import java.text.*;
import java.io.*;

public class FenAppInternational extends java.awt.Frame
{
    private String langueChoisie;
    private static final String NOM_GENERIQUE_FICHIER_PROPERTIES =
        "TextesAeroport";
    private static final String NOM_FICHIER_DESTINATIONS =
        "c:\\java-sun-application\\Internationalisation\\Destinations.txt";

    public FenAppInternational()
    {
        initComponents();
        DialChoixLangue dcl =
            new DialChoixLangue(this, "Aéroport d'Oupeye : choix de la langue", true);
        dcl.setVisible(true);
        langueChoisie = DialChoixLangue.getLangueChoisie();
        System.out.println("Langue de travail choisie : " + langueChoisie);
    }
}

```

```

Locale localeCourant = new Locale
    ((String)DialChoixLangue.codeLangue.get(langueChoisie),
     (String)DialChoixLangue.codePays.get(langueChoisie));

ResourceBundle res =
ResourceBundle.getBundle(NOM_GENERIQUE_FICHIER_PROPERTIES,
    localeCourant);

ZTNom.setText(res.getString("InviteNom"));
ZTDateEnregistrement.setText(res.getString("InfoDateEnregistrement"));
Date maintenant = new Date();
String maDate = DateFormat.
    getDateTimeInstance(DateFormat.FULL,DateFormat.FULL,localeCourant).
    format(maintenant);
ZTDateEnregistrementData.setText(maDate);
ZTDestination.setText(res.getString("InviteDestination"));
ZTDateDepart.setText(res.getString("InviteDateDépart"));

Calendar c = Calendar.getInstance();
for (int i=0; i<70; i++)
{
    c.add(Calendar.HOUR, 24);
    System.out.println(c.getTime());
    String dateSuivante = String.valueOf(c.get(Calendar.DAY_OF_MONTH)) + "/" +
        String.valueOf(c.get(Calendar.MONTH)+1) + "/" +
        String.valueOf(c.get(Calendar.YEAR));
    CBDatesDepart.add(dateSuivante);
}

BOk.setLabel(res.getString("TexteBoutonOk"));
BAnnuler.setLabel(res.getString("TexteBoutonAnnuler"));
setTitle(res.getString("TitreReservation"));

try
{
    BufferedReader fin = new BufferedReader(new FileReader
        (NOM_FICHIER_DESTINATIONS));
    String ligne;
    while ( (ligne=fin.readLine()) != null)
    {
        CBDestinations.add(ligne);
    }
}
catch (FileNotFoundException e)
{ System.out.println("Aïe aie : " + e.getMessage()); }
catch (IOException e)
{ System.out.println("Oie oie : " + e.getMessage()); }
}

private void initComponents() { ... }

```

```
private void exitForm(java.awt.event.WindowEvent evt) { System.exit(0); }

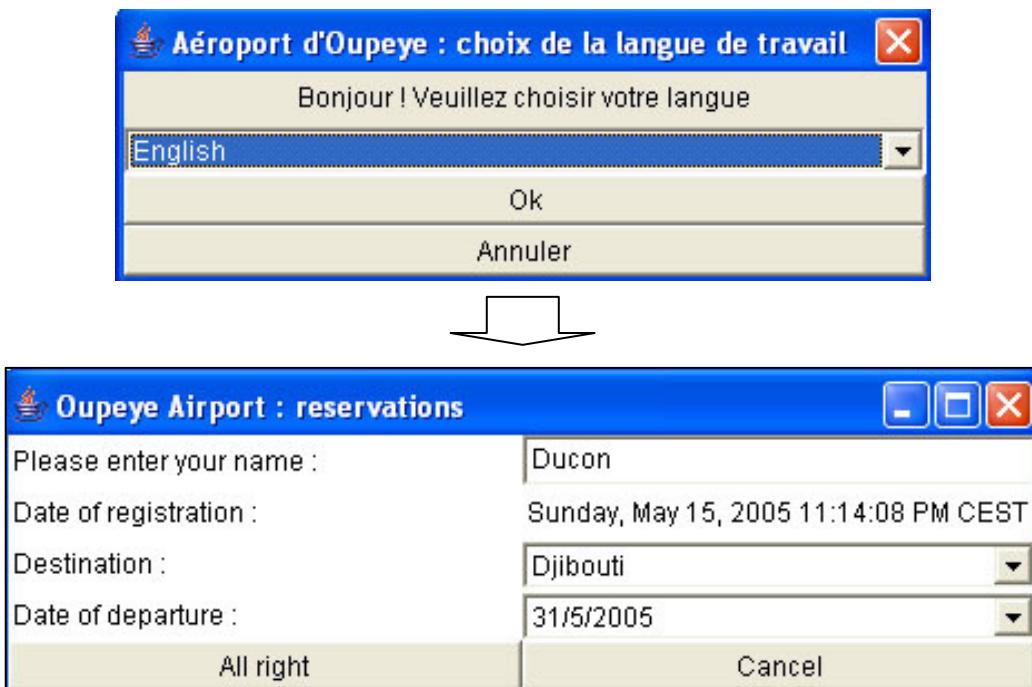
public static void main(String args[])
{
    new FenAppInternational().show();
}

private java.awt.Choice CBDatesDepart;
private java.awt.Label ZTDestination;
private java.awt.TextField textField1;
private java.awt.Choice CBDestinations;
private java.awt.Button BOk;
private java.awt.Label ZTDateDepart;
private java.awt.Label ZTDateEnregistrementData;
private java.awt.Button BAnnuler;
private java.awt.Label ZTDateEnregistrement;
private java.awt.Label ZTNom;
}
```

On remarquera en passant :

- ◆ l'affichage de la date en fonction de l'objet Locale utilisé;
- ◆ le fichier texte des destinations lu avec un Reader;
- ◆ l'utilisation d'un objet Calendar pour afficher les 70 jours suivants la date courante.

Un exemple d'exécution est tout à fait convaincant :



2. Le Collections Framework

2.1 La STL de Java

Nous y avons déjà brièvement fait allusion dans le chapitre VII : le **Java collections framework** peut être comparé à la STL du C++. Il s'agit donc d'une architecture de classes et interfaces visant à proposer au développeur des définitions, des implémentations et des méthodes de manipulation des collections classiques de la programmation : listes, arbres, tables de hashcode avec les opérations de parcours, tris, recherches, etc. L'idée maîtresse est de rester suffisamment abstrait, au sens "éloigné des représentations", pour que les opérations réalisées soient effectuées indépendamment de la nature des objets manipulés.

Viser à l'exhaustivité demanderait probablement un (gros) chapitre entier. Nous allons donc tenter de décrire ici ce qui est le plus couramment utilisé. Pour rappel, le Collections Framework comporte :

- ◆ des interfaces caractérisant les grands types de containers;
- ◆ des implémentations de ces interfaces;
- ◆ des algorithmes qui s'appliquent à ces containers;
- ◆ des itérateurs qui permettent le parcours de ces containers;
- ◆ des comparateurs qui permettent de faire intervenir dans les containers la notion d'ordre;
- ◆ des exceptions spécifiques pour les cas où une opération n'est pas supportée par le container ou lorsque des problèmes d'accès concurrents par itérateurs se posent.

2.2 Les interfaces de base

Comme tout interface, leur rôle est de définir les comportements des différents types de collections envisageables, selon qu'ils sont triés ou pas, qu'ils admettent les doublons ou pas, qu'ils utilisent la notion de clé ou pas. Tous ces interfaces sont des enfants ou des petits-enfants de deux interfaces de base nommés **Collection**, qui caractérise évidemment un container d'objets dont les politiques de tri, d'unicité et d'association sont indéfinies et **Map**, qui caractérise les containers associatifs basés sur la notion de clé, une clé désignant une valeur au plus.

a) Pour **Collection**, les méthodes les plus évocatrices qui y sont déclarées sont :

`public int size()`

Pour fournir le nombre d'éléments de la collection

`public boolean contains (Object o)`

Pour déterminer si un objet se trouve dans la collection ou pas

`public boolean add (Object o)`

Pour bien sûr ajouter un élément à la collection, si du moins la définition de cette collection le permet (autrement dit, cette méthode ne fait rien et retourne false si l'élément se trouve déjà dans la collection et que les doublons sont interdits). En cas d'erreur, plusieurs exceptions sont prévues pour mieux discriminer l'erreur.

`public boolean remove (Object o)`

La même chose, mais à l'envers ;-) ...

`public void clear()`

Evidemment, c'est le grand nettoyage ...

`public Iterator iterator()`

Pour fournir un itérateur pour la collection. Un tel objet, dont le rôle (pour rappel) est de permettre un parcours standardisé d'un container, est une instance d'une classe implémentant l'interface **Iterator**, dont les trois seules méthodes sont :

public boolean hasNext()

Cette méthode devra vérifier si il existe encore un élément sur lequel se déplacer et retournera true en cas de succès, false si il n'y a plus d'éléments.

public Object next()

Cette méthode aura pour rôle de fournir l'élément suivant de la collection,

public void remove()

Si le développeur a choisi de permettre à l'itérateur de retirer un élément de la collection ...

Bien clairement, nous avons ici avec Iterator le successeur de l'interface Enumeration ...

b) Pour Map, les méthodes attendues sont (bien logiquement si on compare avec Hashtable) :

public Object put (Object key, Object value)

public Object get (Object key)

public Object remove (Object key)

2.3 Les interfaces plus spécifiques

a) Dérivés de l'interface de base Collection, les interfaces dédiées à telle ou telle politique de collection sont :

◆ **Set** : correspond à la notion mathématique d'ensemble, c'est-à-dire que la seule restriction est que chaque élément est unique; les méthodes de base sont :

public boolean add(Object o)

sémantique : ajouter un élément à la collection si il n'est pas déjà présent

public boolean remove(Object o)

sémantique : retirer un élément de la collection si il est bien présent

◆ **List** : correspond à la notion mathématique de suite ordonnée (*sequence*), avec la possibilité de se positionner en son sein; les doublons sont à priori permis; les principales méthodes sont :

public boolean add(Object o)

sémantique : ajouter un élément à la fin

public void add(int index, Object element)

sémantique : ajouter un élément à la position indiquée

public Object get(int index)

sémantique : obtenir l'élément se trouvant à la position indiquée

public Object remove(int index)

sémantique : retirer l'élément se trouvant à la position indiquée

avec une spécificité propre aux listes :

public ListIterator listIterator()

L'interface dérivé **ListIterator** apporte en plus les déplacements bidirectionnels et la notion de position, avec des méthodes additionnelles comme

```
public boolean hasPrevious()
public Object previous()
public int nextIndex()
```

Retourne la position de l'élément qui sera renvoyé par le prochain next(). En fait, cette position se trouve toujours entre deux éléments successifs et donc non pas sur un élément donné; si la collection comporte n éléments, il y a n+1 valeurs possibles pour cette position-index :

0	élément[0]	1	élément[1]	2	élément[2]	3	...	n	élément[n]	n+1
↑										

Ceci mis à part, on retrouve donc ici, sous une forme limitée, les différents types d'itérateurs tels que définis dans la STL du C++.

- ◆ **SortedSet** : dérivé de Set, cet interface caractérise une ensemble trié, et déclare logiquement les méthodes :

```
public Object first()
public Object last()
```

La notion de tri est définie

- soit par un ordre implicite basé sur le fait que les objets qu'il contient implémentent l'interface **Comparable** (du package java.lang) dont la seule méthode est

```
public int compareTo (Object o)      // <0, =0 ou >0
```

- soit par un ordre décrit explicitement lors de la création du SortedSet en fournissant un objet implémentant l'interface **Comparator** (du package java.util) qui comporte les deux méthodes

```
public int compare (Object o1, Object o2)
    // comparaison C-like des deux objets : <0, =0 ou >0
public boolean equals (Object obj)
    // comparaison de deux Comparator
```

b) En ce qui concerne l'interface de base **Map**, signalons l'interface :

- ◆ **SortedMap** : c'est évidemment l'équivalent du SortedSet mais pour les Map (il dérive d'ailleurs de ce dernier).

2.4 Les classes de base "semi-abstraites"

L'un ou l'autre des interfaces définis ci-dessus peuvent évidemment être implémentés par un container quelconque créé par le développeur. Cependant, celui-ci appréciera de trouver le travail fait pour les containers courants.

a) Ainsi, dans le domaine des containers non associatifs, on trouve pour débuter la classe :

```
public abstract class AbstractCollection implements Collection
```

qui représente une implémentation minimale quasi-complète d'une collection. La plupart de ses méthodes sont les implémentations des méthodes déclarées dans l'interface Collection, comme :

public boolean add (Object o)

Permet bien sûr d'ajouter un élément à la collection, si du moins la définition de cette collection le permet (autrement dit, cette méthode ne fait rien et retourne false si l'élément se trouve déjà dans la collection et que les doublons sont interdits). En cas d'erreur, plusieurs exceptions sont prévues pour mieux discriminer l'erreur.

public boolean remove (Object o)

La même chose, mais à l'envers ;-)

public void clear()

Vide le container...

public boolean contains (Object o)

Détermine si un objet se trouve dans la collection ou pas selon l'algorithme
o==null ? élément_lu==null : o.equals(élément_lu)

Restent deux méthodes abstraites :

public abstract int size()

Fournit le nombre d'éléments de la collection

public abstract Iterator iterator()

Fournit un itérateur pour la collection.

b) De même, dans le domaine des containers associatifs, on trouve pour débuter la classe :

public abstract class AbstractMap implements Map

qui représente une implémentation minimale quasi-complète d'un container associatif. La plupart de ses méthodes sont les implémentations des méthodes déclarées dans l'interface Map, comme :

public int size()

public Object put (Object key, Object value)

public Object get (Object key)

public Object remove (Object key)

Il reste une méthode abstraite :

public abstract Set entrySet()

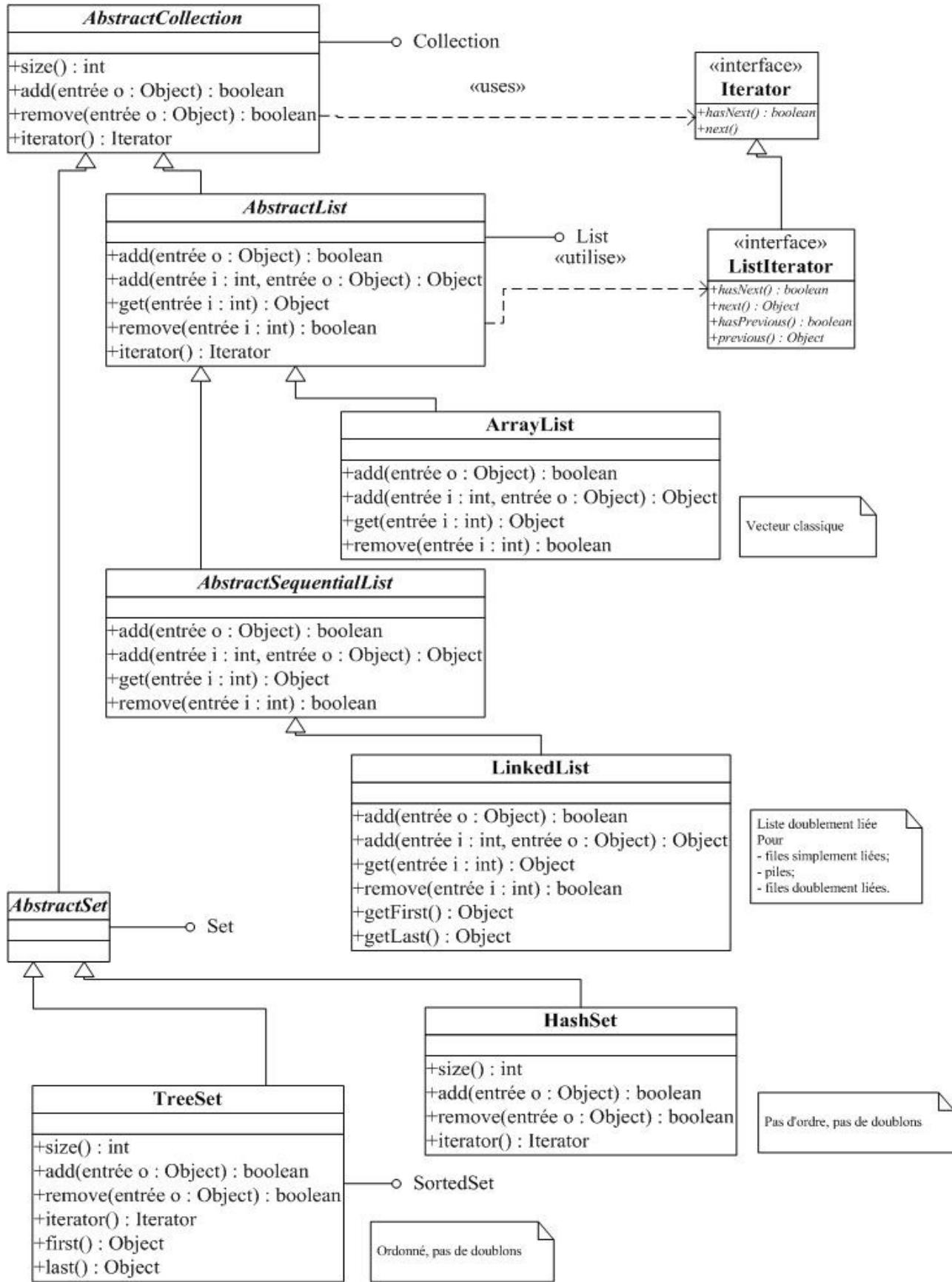
mais elle est de taille puisqu'elle doit fournir l'ensemble sous-jacent contenant les paires (clé, valeur) - de telles paires sont en fait des implémentations de l'interface **Map.Entry**.

2.5 Les classes containers courantes de type Collection

A partir de la classe "semi-abstraite" **AbstractCollection**, il suffit en fait (au minimum, bien sûr) de redéfinir dans une classe dérivée

- ◆ pour une classe au contenu figé : les méthodes iterator() et size()
- ◆ pour une classe instanciée par des objets modifiables : la méthode add() ainsi que la méthode remove() pour l'itérateur associé;

La classe de base **AbstractCollection** sert évidemment de base à une hiérarchie de containers somme toute assez prévisible :



- ◆ **LinkedList** : une liste doublement liée, qui permet donc d'implémenter les classiques files, piles et files doublement liées; on trouve les méthodes typiques :

```
public Object getFirst()
public Object getLast()
public Object removeFirst()
public Object removeLast()
```

- ◆ **HashSet** : un ensemble classique, donc non ordonné, dont les éléments sont retrouvés en interne au moyen d'une hashtable; l'ordre dans lequel on recueille les éléments par itération et donc quelconque et variable au cours du temps.

- ◆ **TreeSet** : un ensemble ordonné de type SortedSet; selon la manière dont la relation d'ordre entre les éléments est implémentée, on utilise l'un des constructeurs :

```
public TreeSet()
public TreeSet (Comparator c)
```

2.6 Un exemple élémentaire d'utilisation des collections

Afin d'illustrer sommairement ces différentes collections, nous allons créer des containers de divers types contenant tous des contrats d'assurance voiture instances de la classe :

ContratAssuranceVoiture.java

```
/*
 * ContratAssuranceVoiture.java
 * Created on 1 juin 2005, 9:11
 */
/**
 * @author Vilvens
 */

public class ContratAssuranceVoiture implements Comparable
{
    protected String numeroPolice;
    protected String numeroPlaque;
    protected String nomClient;

    public ContratAssuranceVoiture(String nPo, String nPl, String n)
    {
        numeroPolice = nPo;
        numeroPlaque = nPl;
        nomClient = n;
    }

    public String getNumeroPolice() { return numeroPolice; }
    public String getNumeroPlaque() { return numeroPlaque; }
    public String getNomClient() { return nomClient; }
```

```

public String toString()
{
    return new String ("Plaque : " + getNumeroPlaque() + " - pol. "
        + getNumeroPolice() + " --> " + getNomClient());
}

public boolean equals(ContratAssuranceVoiture c)
{
    System.out.println("$$$$$$$$$ Passage par equals maison ");
    if (numeroPolice.equals(c.getNumeroPolice())) return true;
    else return false;
}

public int compareTo (Object c)
{
    if (getClass().getName().equals("ContratAssuranceVoiture"))
        return numeroPolice.compareTo(((ContratAssuranceVoiture)c).getNumeroPolice());
    else return compareTo(c);
}

public int hashCode()
{
    System.out.println("$+$$$$$$ Passage par hashCode maison :"
        + numeroPolice + " --> "
        + numeroPolice.hashCode());
    return numeroPolice.hashCode();
}
}

```

On peut remarquer :

- ◆ la redéfinition de la méthode **equals()**;
- ◆ celle de la méthode **hashcode()**;
- ◆ l'implémentation de l'interface **Comparable** qui implique la définition de la méthode **compareTo()** – elle explique comment comparer par défaut deux contrats d'assurance.

On peut alors imaginer la petite application suivante qui crée toute une série de containers à partir d'un tableau d'objets ContratAssuranceVoiture :

ContainersContratsAssuranceVoiture.java

```

/*
 * ContainersContratsAssuranceVoiture.java
 * Created on 1 juin 2005, 9:15
 */
/**
 * @author Vilvens
 */

import java.util.*;

```

```

public class ContainersContratsAssuranceVoiture
{
    public static void main(String[] args)
    {
        ContratAssuranceVoiture c;

        ContratAssuranceVoiture[] tabContrats =
        { new ContratAssuranceVoiture("125487", "ETJ988", "van Pieperseele"),
          new ContratAssuranceVoiture("454663", "DXR599", "Beauchassis"),
          new ContratAssuranceVoiture("007007", "M6007", "Bond"),
          new ContratAssuranceVoiture("008008", "XYZ123", "Vil"),
          (c=new ContratAssuranceVoiture("007007", "M60072", "St Clair")),
          c,
          new ContratAssuranceVoiture("526398", "RFC822", "Carcler")
        };

        if (tabContrats[2].equals(tabContrats[4])) System.out.println("Egaux");
        else System.out.println("PAS égaux");

ArrayList al = new ArrayList();
testCollection (al, tabContrats,
                 new ContratAssuranceVoiture("774021", "NOC541", "Letueur"));

LinkedList ll = new LinkedList();
testCollection (ll, tabContrats,
                 new ContratAssuranceVoiture("774021", "NOC541", "Letueur"));
afficheListReverse(ll, "* Dans l'ordre inverse *");
System.out.println("On retire : " + ll.removeFirst());
afficheCollection(ll, "- Après removeFirst");

Vector v = new Vector();
testCollection (v, tabContrats,
                 new ContratAssuranceVoiture("774021", "NOC541", "Letueur"));

Stack p = new Stack();
testCollection (p, tabContrats,
                 new ContratAssuranceVoiture("774021", "NOC541", "Letueur"));
p.push(new ContratAssuranceVoiture("696969", "GRR999", "Ladentdure"));
afficheCollection(p, "- Après un push");
p.pop();
afficheCollection(p, "- Après un pop");

HashSet hs = new HashSet();
testCollection (hs, tabContrats,
                 new ContratAssuranceVoiture("774021", "NOC541", "Letueur"));

TreeSet ts = new TreeSet();
testCollection (ts, tabContrats,
                 new ContratAssuranceVoiture("774021", "NOC541", "Letueur"));

```

```

TreeSet tsc = new TreeSet(new CompareContratAssuranceVoiture());
testCollection (tsc, tabContrats,
    new ContratAssuranceVoiture("774021", "NOC541", "Letueur"));
}

public static void testCollection (Collection c, ContratAssuranceVoiture[] t,
    ContratAssuranceVoiture no)
{
    afficheCollection(c, "Contenu initial");
    ajouteCollection(c, t);
    afficheCollection(c, "Après ajout massif");
    c.remove(t[2]); c.remove(t[0]);
    afficheCollection(c, "Après retrait t[2] et t[0]");
    c.add(no);
    afficheCollection(c, "Après ajout");
}

public static void afficheCollection (Collection c)
{
    if (c.isEmpty()) System.out.println("Container vide");
    else System.out.println("Nombre d'éléments = " + c.size());
    Iterator it = c.iterator();
    while (it.hasNext())
    {
        System.out.println(it.next());
    }

    System.out.println(" [Type de container utilisé : " + c.getClass().getName() + "]");
}

public static void afficheCollection (Collection c, String titre)
{
    System.out.println("--" + titre + "--");
    afficheCollection(c);
}

public static void afficheListReverse (List c)
{
    if (c.isEmpty()) System.out.println("Container vide");
    else System.out.println("Nombre d'éléments = " + c.size());
    ListIterator it = c.listIterator(c.size());
    while (it.hasPrevious())
    {
        System.out.println(it.previous());
    }

    System.out.println(" [Type de container utilisé : " +
        c.getClass().getName() + "]");
}

```

```

public static void afficheListReverse (List c, String titre)
{
    System.out.println("--" + titre + "--");
    afficheListReverse(c);
}

public static void ajouteCollection (Collection c, ContratAssuranceVoiture[] t)
{
    for (int i=0; i<t.length; i++)
    {
        if (c.contains(t[i])) System.out.println(" Elément à ajouter déjà présent");
        else System.out.println(" Elément à ajouter non encore présent");
        c.add(t[i]);
    }
}
}

```

On remarquera encore le deuxième TreeSet qui utilise un Comparator défini par :

CompareContratAssuranceVoiture.java

```

/*
 * CompareContratAssuranceVoiture.java
 * Created on 14 juin 2005, 15:53
 */
/**
 * @author Vilvens
 */
import java.util.*;

public class CompareContratAssuranceVoiture implements Comparator
{
    public CompareContratAssuranceVoiture() { }

    public int compare (Object o1, Object o2)
    {
        if (o1.getClass().getName().equals("ContratAssuranceVoiture"))
            return ((ContratAssuranceVoiture)o1).getNumeroPlaque();
            compareTo(((ContratAssuranceVoiture)o2).getNumeroPlaque());
        else return -1;
    }
}

```

- celui-ci compare donc les contrats sur base des plaques de voiture au lieu des numéros de police.

Les affichages console que l'on obtient sont du type suivant :

\$\$\$\$\$\$\$\$ Passage par equals maison Egaux

--Contenu initial--

Container vide

[Type de container utilisé : java.util.ArrayList]

Elément à ajouter non encore présent

Elément à ajouter déjà présent

Elément à ajouter non encore présent

--Après ajout massif--

Nombre d'éléments = 7

Plaque : ETJ988 - pol. 125487 --> van Pieperseele

Plaque : DXR599 - pol. 454663 --> Beauchassis

Plaque : M6007 - pol. 007007 --> Bond

Plaque : XYZ123 - pol. 008008 --> Vil

Plaque : M60072 - pol. 007007 --> St Clair

Plaque : M60072 - pol. 007007 --> St Clair

Plaque : RFC822 - pol. 526398 --> Carcler

[Type de container utilisé : java.util.ArrayList]

--Après retrait t[2] et t[0]--

Nombre d'éléments = 5

Plaque : DXR599 - pol. 454663 --> Beauchassis

Plaque : XYZ123 - pol. 008008 --> Vil

Plaque : M60072 - pol. 007007 --> St Clair

Plaque : M60072 - pol. 007007 --> St Clair

Plaque : RFC822 - pol. 526398 --> Carcler

[Type de container utilisé : java.util.ArrayList]

--Après ajout--

Nombre d'éléments = 6

Plaque : DXR599 - pol. 454663 --> Beauchassis

Plaque : XYZ123 - pol. 008008 --> Vil

Plaque : M60072 - pol. 007007 --> St Clair

Plaque : M60072 - pol. 007007 --> St Clair

Plaque : RFC822 - pol. 526398 --> Carcler

Plaque : NOC541 - pol. 774021 --> Letueur

[Type de container utilisé : java.util.ArrayList]

--Contenu initial--

Container vide

[Type de container utilisé : java.util.LinkedList]

Elément à ajouter non encore présent

...

Elément à ajouter déjà présent

Elément à ajouter non encore présent

dans ajouteCollection : C[0] est contenu

--Après ajout massif--

Nombre d'éléments = 7

Plaque : ETJ988 - pol. 125487 --> van Pieperseele

...

Plaque : RFC822 - pol. 526398 --> Carcler
[Type de container utilisé : **java.util.LinkedList**]
--Après retrait t[2] et t[0]--
Nombre d'éléments = 5
Plaque : DXR599 - pol. 454663 --> Beauchassis
Plaque : XYZ123 - pol. 008008 --> Vil
Plaque : M60072 - pol. 007007 --> St Clair
Plaque : M60072 - pol. 007007 --> St Clair
Plaque : RFC822 - pol. 526398 --> Carcler
[Type de container utilisé : **java.util.LinkedList**]
--Après ajout--
Nombre d'éléments = 6
Plaque : DXR599 - pol. 454663 --> Beauchassis
Plaque : XYZ123 - pol. 008008 --> Vil
Plaque : M60072 - pol. 007007 --> St Clair
Plaque : M60072 - pol. 007007 --> St Clair
Plaque : RFC822 - pol. 526398 --> Carcler
Plaque : NOC541 - pol. 774021 --> Letueur
[Type de container utilisé : **java.util.LinkedList**]
--* **Dans l'ordre inverse***--
Nombre d'éléments = 6
Plaque : NOC541 - pol. 774021 --> Letueur
Plaque : RFC822 - pol. 526398 --> Carcler
Plaque : M60072 - pol. 007007 --> St Clair
Plaque : M60072 - pol. 007007 --> St Clair
Plaque : XYZ123 - pol. 008008 --> Vil
Plaque : DXR599 - pol. 454663 --> Beauchassis
[Type de container utilisé : **java.util.LinkedList**]
On retire : Plaque : DXR599 - pol. 454663 --> Beauchassis
--- **Apres removeFirst**--
Nombre d'éléments = 5
Plaque : XYZ123 - pol. 008008 --> Vil
Plaque : M60072 - pol. 007007 --> St Clair
Plaque : M60072 - pol. 007007 --> St Clair
Plaque : RFC822 - pol. 526398 --> Carcler
Plaque : NOC541 - pol. 774021 --> Letueur
[Type de container utilisé : **java.util.LinkedList**]

--Contenu initial--
Container vide
[Type de container utilisé : **java.util.Vector**]
Elément à ajouter non encore présent
...
Elément à ajouter déjà présent
Elément à ajouter non encore présent
--Après ajout massif--
Nombre d'éléments = 7
Plaque : ETJ988 - pol. 125487 --> van Pieperseele
...
[Type de container utilisé : **java.util.Vector**]

...

--Contenu initial--

Container vide

[Type de container utilisé : java.util.Stack]

Elément à ajouter non encore présent

...

Elément à ajouter déjà présent

Elément à ajouter non encore présent

--Après ajout massif--

Nombre d'éléments = 7

Plaque : ETJ988 - pol. 125487 --> van Pieperseele

...

Plaque : RFC822 - pol. 526398 --> Carcler

[Type de container utilisé : **java.util.Stack**]

...

-- Après un push--

Nombre d'éléments = 7

Plaque : DXR599 - pol. 454663 --> Beauchassis

Plaque : XYZ123 - pol. 008008 --> Vil

Plaque : M60072 - pol. 007007 --> St Clair

Plaque : M60072 - pol. 007007 --> St Clair

Plaque : RFC822 - pol. 526398 --> Carcler

Plaque : NOC541 - pol. 774021 --> Letueur

Plaque : GRR999 - pol. 696969 --> Ladendure

[Type de container utilisé : java.util.Stack]

-- Après un pop--

Nombre d'éléments = 6

Plaque : DXR599 - pol. 454663 --> Beauchassis

Plaque : XYZ123 - pol. 008008 --> Vil

Plaque : M60072 - pol. 007007 --> St Clair

Plaque : M60072 - pol. 007007 --> St Clair

Plaque : RFC822 - pol. 526398 --> Carcler

Plaque : NOC541 - pol. 774021 --> Letueur

[Type de container utilisé : java.util.Stack]

--Contenu initial--

Container vide

[Type de container utilisé : java.util.HashSet]

\$+\$\$\$\$\$\$ Passage par hashCode maison :125487 --> 1450635135

Elément à ajouter non encore présent

\$+\$\$\$\$\$\$ Passage par hashCode maison :125487 --> 1450635135

\$+\$\$\$\$\$\$ Passage par hashCode maison :454663 --> 1539265216

...

Elément à ajouter déjà présent

\$+\$\$\$\$\$\$ Passage par hashCode maison :007007 --> 1420214432

\$+\$\$\$\$\$\$ Passage par hashCode maison :526398 --> 1565180601

...

--Après ajout massif--

Nombre d'éléments = 6

Plaque : RFC822 - pol. 526398 --> Carcler

Plaque : M60072 - pol. 007007 --> St Clair
Plaque : M6007 - pol. 007007 --> Bond
Plaque : DXR599 - pol. 454663 --> Beauchassis
Plaque : XYZ123 - pol. 008008 --> Vil
Plaque : ETJ988 - pol. 125487 --> van Pieperseele
[Type de container utilisé : **java.util.HashSet**]
\$+\$\$\$\$\$\$ Passage par hashCode maison :007007 --> 1420214432
\$+\$\$\$\$\$\$ Passage par hashCode maison :125487 --> 1450635135
--Après retrait t[2] et t[0]--
Nombre d'éléments = 4
Plaque : RFC822 - pol. 526398 --> Carcler
Plaque : M60072 - pol. 007007 --> St Clair
Plaque : DXR599 - pol. 454663 --> Beauchassis
Plaque : XYZ123 - pol. 008008 --> Vil
[Type de container utilisé : **java.util.HashSet**]
\$+\$\$\$\$\$\$ Passage par hashCode maison :774021 --> 1626993819
--Après ajout--
Nombre d'éléments = 5
Plaque : RFC822 - pol. 526398 --> Carcler
Plaque : M60072 - pol. 007007 --> St Clair
Plaque : DXR599 - pol. 454663 --> Beauchassis
Plaque : NOC541 - pol. 774021 --> Letueur
Plaque : XYZ123 - pol. 008008 --> Vil
[Type de container utilisé : **java.util.HashSet**]

--Contenu initial--
Container vide
[Type de container utilisé : **java.util.TreeSet**]
Elément à ajouter non encore présent
Elément à ajouter déjà présent
Elément à ajouter déjà présent
Elément à ajouter non encore présent
dans ajouteCollection : C[0] est contenu
--Après ajout massif--
Nombre d'éléments = 5
Plaque : **M6007 - pol. 007007 --> Bond**
Plaque : **XYZ123 - pol. 008008 --> Vil**
Plaque : **ETJ988 - pol. 125487 --> van Pieperseele**
Plaque : **DXR599 - pol. 454663 --> Beauchassis**
Plaque : **RFC822 - pol. 526398 --> Carcler**
[Type de container utilisé : **java.util.TreeSet**]
--Après retrait t[2] et t[0]--
Nombre d'éléments = 3
Plaque : XYZ123 - pol. 008008 --> Vil
Plaque : DXR599 - pol. 454663 --> Beauchassis
Plaque : RFC822 - pol. 526398 --> Carcler
[Type de container utilisé : **java.util.TreeSet**]

--Après ajout--

Nombre d'éléments = 4

Plaque : XYZ123 - pol. 008008 --> Vil

Plaque : DXR599 - pol. 454663 --> Beauchassis

Plaque : RFC822 - pol. 526398 --> Carcler

Plaque : NOC541 - pol. 774021 --> Letueur

[Type de container utilisé : java.util.TreeSet]

--Contenu initial--

Container vide

[Type de container utilisé : java.util.TreeSet]

Elément à ajouter non encore présent

...

Elément à ajouter non encore présent

Elément à ajouter déjà présent

Elément à ajouter non encore présent

dans ajouteCollection : C[0] est contenu

--Après ajout massif--

Nombre d'éléments = 6

Plaque : DXR599 - pol. 454663 --> Beauchassis

Plaque : ETJ988 - pol. 125487 --> van Piepersele

Plaque : M6007 - pol. 007007 --> Bond

Plaque : M60072 - pol. 007007 --> St Clair

Plaque : RFC822 - pol. 526398 --> Carcler

Plaque : XYZ123 - pol. 008008 --> Vil

[Type de container utilisé : java.util.TreeSet]

--Après retrait t[2] et t[0]--

Nombre d'éléments = 4

Plaque : DXR599 - pol. 454663 --> Beauchassis

Plaque : M60072 - pol. 007007 --> St Clair

Plaque : RFC822 - pol. 526398 --> Carcler

Plaque : XYZ123 - pol. 008008 --> Vil

[Type de container utilisé : java.util.TreeSet]

--Après ajout--

Nombre d'éléments = 5

Plaque : DXR599 - pol. 454663 --> Beauchassis

Plaque : M60072 - pol. 007007 --> St Clair

Plaque : NOC541 - pol. 774021 --> Letueur

Plaque : RFC822 - pol. 526398 --> Carcler

Plaque : XYZ123 - pol. 008008 --> Vil

[Type de container utilisé : java.util.TreeSet]

Ceci nous montre tout de même que :

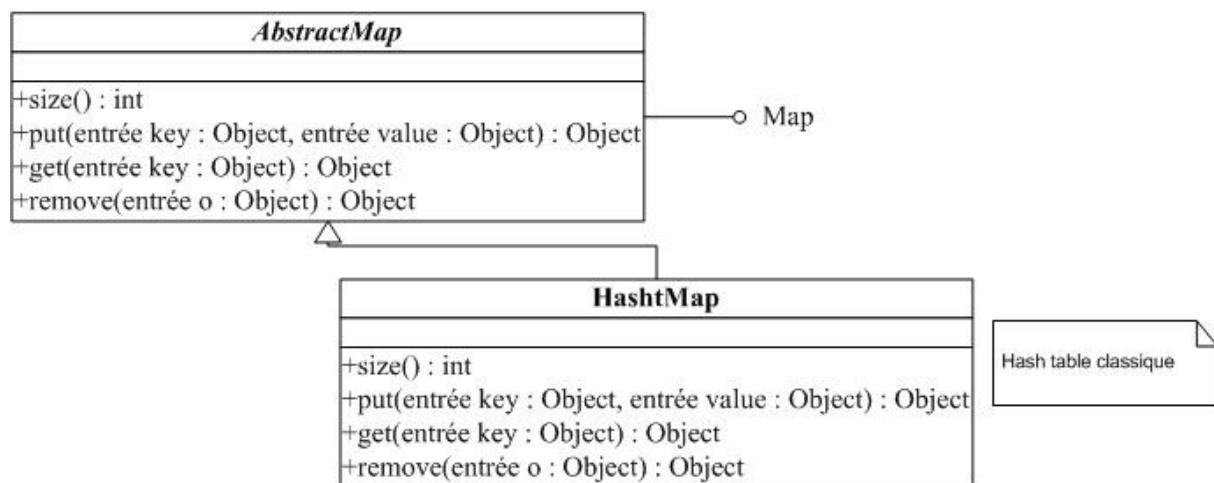
- ◆ **notre méthode equals() est totalement ignorée par les containers admettant les doublons** ! En fait, ils utilisent la méthode equals() d'Object, ce qui explique que le seul objet assurance détecté en double est le contrat "c";
 - ◆ la deuxième version du TreeSet a bien trié selon les numéros de plaque – du coup, il contient alors 6 éléments au lieu de 5 comme dans la première version.
-

2.7 Les classes containers courantes de type Map

A partir de la classe "semi-abstraite" **AbstractMap**, il suffit en fait (au minimum, bien sûr) de redéfinir dans une classe dérivée

- ◆ pour une classe au contenu figé : la méthode entrySet();
- ◆ pour une classe instanciée par des objets modifiables : la méthode put() ainsi que la méthode remove() pour l'itérateur associé à l'ensemble que fournit entrySet().

Il existe donc une deuxième hiérarchie basée sur la classe **AbstractMap**, mais nous n'en retiendrons qu'une classe dérivée :



- ◆ **HashMap** : c'est la hashtable classique, avec le support de clés ou de valeurs null; c'est la version non synchronized de la classe Hastable classique; on retrouve donc les méthodes des containers associatifs :

```
public Object put (Object key, Object value)
public Object get (Object key)
```

- ce qui ne change pas grand' chose à nos habitudes ...

2.8 Les algorithmes classiques

La classe **Collections** du package `java.util` ne contient que des méthodes statiques : en fait, on en dirait que ce sont des fonctions utilitaires dans d'autres langages comme C et C++. Un grand nombre d'entre elles s'applique à des containers de type `List`, comme par exemple :

```
public static void reverse (List list)
    Place les éléments de la liste dans l'ordre inverse
public static void sort (List list)
    Trie les éléments de la liste
public static int binarySearch (List list, Object key)
```

On peut imaginer, dans la foulée du programme précédent :

```
...
LinkedList ll = new LinkedList();
testCollection (ll, tabContrats,
    new ContratAssuranceVoiture("774021", "NOC541", "Letueur"));
afficheListReverse(ll, "* Dans l'ordre inverse *");
System.out.println("On retire : " + ll.removeFirst());
afficheCollection(ll, "- Apres removeFirst");
Collections.sort(ll);
afficheCollection(ll, "----- Apres un tri");
Collections.reverse(ll);
afficheCollection(ll, "----- Apres une inversion");
...

public static void afficheListReverse (List c)
{
    if (c.isEmpty()) System.out.println("Container vide");
    else System.out.println("Nombre d'éléments = " + c.size());
    ListIterator it = c.listIterator(c.size());
    while (it.hasPrevious())
    {
        System.out.println(it.previous());
    }

    System.out.println(" [Type de container utilisé : " + c.getClass().getName() + "]");
}
```

ce qui donne :

```
Nombre d'éléments = 6
Plaque : DXR599 - pol. 454663 --> Beauchassis
Plaque : XYZ123 - pol. 008008 --> Vil
Plaque : M60072 - pol. 007007 --> St Clair
Plaque : M60072 - pol. 007007 --> St Clair
Plaque : RFC822 - pol. 526398 --> Carcler
Plaque : NOC541 - pol. 774021 --> Letueur
--* Dans l'ordre inverse --
Nombre d'éléments = 6
Plaque : NOC541 - pol. 774021 --> Letueur
Plaque : RFC822 - pol. 526398 --> Carcler
Plaque : M60072 - pol. 007007 --> St Clair
Plaque : M60072 - pol. 007007 --> St Clair
Plaque : XYZ123 - pol. 008008 --> Vil
Plaque : DXR599 - pol. 454663 --> Beauchassis
    [Type de container utilisé : java.util.LinkedList]
On retire : Plaque : DXR599 - pol. 454663 --> Beauchassis
-- Apres removeFirst --
Nombre d'éléments = 5
Plaque : XYZ123 - pol. 008008 --> Vil
Plaque : M60072 - pol. 007007 --> St Clair
```

```
Plaque : M60072 - pol. 007007 --> St Clair
Plaque : RFC822 - pol. 526398 --> Carcler
Plaque : NOC541 - pol. 774021 --> Letueur
[Type de container utilisé : java.util.LinkedList]
```

----- **Apres un tri --**

```
Nombre d'éléments = 5
Plaque : M60072 - pol. 007007 --> St Clair
Plaque : M60072 - pol. 007007 --> St Clair
Plaque : XYZ123 - pol. 008008 --> Vil
Plaque : RFC822 - pol. 526398 --> Carcler
Plaque : NOC541 - pol. 774021 --> Letueur
[Type de container utilisé : java.util.LinkedList]
```

----- **Apres une inversion --**

```
Nombre d'éléments = 5
Plaque : NOC541 - pol. 774021 --> Letueur
Plaque : RFC822 - pol. 526398 --> Carcler
Plaque : XYZ123 - pol. 008008 --> Vil
Plaque : M60072 - pol. 007007 --> St Clair
Plaque : M60072 - pol. 007007 --> St Clair
[Type de container utilisé : java.util.LinkedList]
```

Le tri a donc été réalisé sur base de l'implémentation de Comparable, donc par numéro de police. Si l'on voulait utiliser un autre critère de comparaison, on utiliserait

```
public static void sort (List list, Comparator c)
```

Quelques méthodes de Collections s'appliquent cependant aux collections générales, comme :

```
public static Object min (Collection coll)
public static Object max (Collection coll)
```

Fournissent évidemment les minimum et maximum du container

```
public static Object min (Collection coll, Comparator comp)
public static Object max (Collection coll, Comparator comp)
```

Les mêmes, mais avec un critère de comparaison différent de celui utilisé implicitement.

Avec la liste utilisée précédemment, on peut donc programmer :

```
...
System.out.println("Valeur minimale de la liste (polices) : " + Collections.min(ll));
System.out.println("Valeur minimale de la liste (plaques) : " +
    Collections.max(ll, new CompareContratAssuranceVoiture()));
...
```

pour obtenir :

```
Valeur minimale de la liste (polices) : Plaque : M60072 - pol. 007007 --> St Clair
Valeur minimale de la liste (plaques) : Plaque : XYZ123 - pol. 008008 --> Vil
```

2.9 Les évolutions génériques du JDK 1.5

Les containers décrits ci-dessus se déclinent, depuis le JDK 1.5, en version **générique** (un programmeur C++ aurait dit "*template*"), c'est-à-dire que chacune de ces classes est instanciable en précisant la nature des objets qu'elle va contenir. On peut encore parler de "classes paramétrées". Ceci permet un meilleur contrôle de la nature des objets que l'on tente d'y placer et dispense également des castings lors de la lecture des éléments.

La syntaxe est manifestement inspirée de celle du C++. Ainsi, un vecteur (c'est-à-dire une instance de Vector) destiné à ne contenir que des chaînes de caractères se déclarera par :

```
Vector<String> vs = new Vector<String>();
```

On y placera des éléments instances de la classe précisée à la définition (donc, ici, des instances de String), par :

```
vs.add("Vilvens");vs.add("Charlet");vs.add("Mercenier");
```

Parcourir un tel vecteur peut se faire au moyen d'une boucle for apparue également dans le JDK 1.5 et qui ne réclame plus la description détaillée de l'itération lorsqu'elle s'applique à une collection – on peut encore parler de "boucle for à description implicite" :

```
for (String i:vs)  
    System.out.println(i);
```

Enfin, on peut évidemment définir dans la foulée un vecteur destiné à contenir des entiers :

```
Vector<Integer> vi = new Vector<Integer>();
```

A priori, personne ne sera surpris de voir la classe Integer utilisée à la place du type primitif int. Cependant, des instructions comme :

```
vi.add(25); vi.add(-36); vi.add(57);
```

ne poseront, elles non plus, aucun problème ! En fait, les valeurs entières de type primitif ont été automatiquement transformées en des instances de la classe Intger : on parle encore d"**"autoboxing"** (et évidemment d"**"unboxing"** dans l'autre sens).

Le petit programme suivant illustre ces possibilité avec un vecteur supplémentaire d'objets ContratAssuranceVoiture et une méthode affiche() à paramètre template :

TestTemplate.java

```
/*  
 * TestTemplate.java  
 *  
 */  
package templates15;  
  
/**  
 * @author Vilvens  
 */
```

```
import java.util.*;  
  
public class TestTemplate<T>  
{  
    public TestTemplate() {}  
  
    public static void main(String[] args)  
    {  
        System.out.println("1ère salve");  
  
        Vector<String> vs = new Vector<String>();  
        vs.add("Vilvens");vs.add("Charlet");vs.add("Mercenier");  
        vs.add("Madani");vs.add("Kuty");vs.add("Wagner");  
  
        for (String i:vs)  
            System.out.println(i);  
  
        int [] array = new int [] { 25, -36, 57, -71, 85 };  
        for( int i : array )  
            System.out.println(i);  
  
        Vector<Integer> vi = new Vector<Integer>();  
        vi.add(25); vi.add(-36);vi.add(57);vi.add(-71);vi.add(85);  
  
        for (int i:vi)  
            System.out.println(i);  
  
        Vector<ContratAssuranceVoiture > vcav = new Vector<ContratAssuranceVoiture >();  
        vcav.add(new ContratAssuranceVoiture("55646", "ETJ988", "Herniet"));  
        vcav.add(new ContratAssuranceVoiture("78546", "DXR599", "Cleront"));  
        vcav.add(new ContratAssuranceVoiture("632846", "ANN923", "Thity"));  
  
        for (ContratAssuranceVoiture i:vcav)  
            System.out.println(i);  
  
        System.out.println("2ème salve");  
        new TestTemplate().affiche(vs);  
        new TestTemplate().affiche(vi);  
        new TestTemplate().affiche(vcav);  
    }  
  
    public void affiche( Vector<T> v)  
    {  
        for ( T x : v)  
            System.out.println(x);  
    }  
}
```

3. Les impressions

L'utilisation de l'imprimante a déjà été évoquée à la fin du chapitre consacré aux flux. Cependant, il ne s'agissait là que d'imprimer ce qui était visible dans une fenêtre graphique (la méthode d'impression se référant à la méthode `paint()` pour savoir ce qui devait être imprimé). Nous allons ici nous préoccuper d'imprimer, par exemple, un texte ou une image contenus dans un fichier – le problème classique.

3.1 L'origine et le format des données à imprimer

Parmi les opérations d'entrées-sorties, l'impression est sans doute l'une de celles qui sont les plus délicates : outre le fait que les périphériques d'impression sont très variés, les formats sous lesquels les informations à imprimer sont présentées sont également nombreux. De plus, il faut pouvoir déterminer si tel format est supporté par tel périphérique. Ceci explique l'existence d'une classe **DocFlavor** (package `javax.print`) dont le rôle est d'encapsuler un objet qui permettra de spécifier le format et la source des données à imprimer. Un tel objet devra donc contenir :

- ◆ un type MIME (**Multipurpose Internet Mail Extensions** – protocole décrivant comment représenter des données de natures diverses de manière à ce qu'elles puissent être utilisées par un système de messagerie électronique ou un browser HTTP);
- ◆ le nom de la classe dont l'objet qui fournit les données à imprimer est une instance.

Par exemple, du texte plat ASCII (le classique et passe-partout "text/plain") obtenu à partir d'un flux classique sera associé à un DocFlavor dont le constructeur

```
public DocFlavor (String mimeType, String className)
```

aura reçu comme arguments "text/plain" et "java.io.InputStream".

La classe DocFlavor contient en fait un certain nombre de classes imbriquées (nested), dérivées de DocFlavor et associées aux classes classiques fournissant des données, qui elles-mêmes contiennent des objets DocFlavor prédéfinis correspondant aux types MIME les plus courants.

a) Ainsi, par exemple, on trouve dans DocFlavor la classe imbriquée

```
public static class DocFlavor.INPUT_STREAM extends DocFlavor
```

qui correspond à des objets fournisseurs qui sont des flux (des `java.io.InputStream`). Au sein de cette classe, on trouve des constantes de classes comme :

```
public static final DocFlavor.INPUT_STREAM TEXT_PLAIN_HOST
    // type MIME "text/plain"
public static final DocFlavor.INPUT_STREAM TEXT_PLAIN_US_ASCII
    // type MIME "text/plain; charset=us-ascii"
public static final DocFlavor.INPUT_STREAM TEXT_HTML_HOST
    // type MIME "text/html"
public static final DocFlavor.INPUT_STREAM JPEG
    // type MIME "image/jpeg"
etc
```

et même

```
public static final DocFlavor.INPUT_STREAM AUTOSENSE
```

qui correspond au type MIME "fourre-tout" "application/octet-stream" ce qui suppose qu'il est supposé que l'imprimante saura quoi faire avec les données reçues.

b) D'autres classes imbriquées correspondent aux sources de données usuelles, comme :

```
public static class DocFlavor.BYTE_ARRAY extends DocFlavor
```

la classe productrice est un byte[] – les constantes de classes exposées ci-dessus sont également disponibles;

```
public static class DocFlavor.URL
```

pour les données en provenance d'une URL – à nouveau, les constantes de classes exposées ci-dessus sont également disponibles;

```
public static class DocFlavor.READER
```

la classe productrice est un java.io.Reader; cette fois, et bien logiquement seules deux constantes de classes sont disponibles :

```
public static final DocFlavor.READER TEXT_PLAIN
```

```
public static final DocFlavor.READER TEXT_HTML
```

3.2 Le service d'impression

L'interface **PrintService** caractérise en fait les classes factory des objets **DocPrintJob** qui seront responsables de l'impression effective (voir paragraphe suivant). Associée à une imprimante donnée, une telle classe est dépositaire des possibilités de celle-ci, notamment des types de documents qu'elle supporte. Un tel objet **PrintService** sera capable de créer effectivement le job d'impression demandé. Dans la version la plus simple, on recherche un tel objet au moyen de la méthode de la classe **PrintServiceLookup** :

```
public static final PrintService lookupDefaultPrintService()
```

qui recherche le service associé à l'imprimante par défaut. Mais il existe une version polymorphe plus générale de cette méthode :

```
public static final PrintService[] lookupPrintServices (DocFlavor flavor, AttributeSet attributes)
```

Celle-ci recherche tous les services associés à un type de données (MIME et source) et à un format d'impression donné – ce format est précisé dans un objet implementant l'interface **AttributeSet** (du package javax.print.attribute). Comme son nom l'indique, il sert de container à des objets implementant l'interface **Attribute** (même package), dont les implémentations décrivent toute une série de caractéristiques d'un document. Sans prétendre à l'exhaustivité, citons parmi ces interfaces enfants d'AttributeSet :

- ◆ **DocAttributeSet**, ensemble d'objets **Attribute** qui sont implementés, par exemple, par la classe **Media** et sa fille **MediaSizeName** (du package javax.print.attribute.standard), qui comporte des constantes indiquant la taille du document à imprimer comme :
-

```
public static final MediaSizeName ISO_A3  
public static final MediaSizeName ISO_A4
```

ou encore par son autre fille **MediaTray** qui comporte des constantes comme

```
public static final MediaTray MANUAL
```

♦ **PrintRequestAttributeSet** qui est implémenté par la classe **HashPrintRequestAttributeSet**, dont le nom indique qu'elle est construite sur une structure d'ensemble géré en table de hachage; celle-ci regroupe des informations analogues à **DocAttributeSet**, mais pour tous les documents placés dans un même job d'impression.

3.3 Le job d'impression

Une fois un objet service obtenu, on peut lui demander de créer un job d'impression qui mènera à bien le processus d'impression. Pour obtenir une instance de job, on utilisera la factory que la classe service doit implémenter :

```
public DocPrintJob createPrintJob()
```

DocPrintJob est en fait un interface dont la méthode à implémenter par l'objet créé qui nous intéresse le plus est

```
public void print(Doc doc, PrintRequestAttributeSet attributes)  
throws PrintException
```

On se doute que le 2^{ème} paramètre permet de spécifier les paramètres d'impression évoqués plus haut. Le 1^{er} paramètre permet bien entendu de spécifier la source de données : celle-ci est représentée par un objet implémentant l'interface **Doc**. Pour ce qui nous concerne, le plus simple est d'utiliser la classe **SimpleDoc** (qui est d'ailleurs final) et dont le constructeur a pour prototype :

```
public SimpleDoc(Object printData, DocFlavor flavor, DocAttributeSet attributes)
```

3.4 Le programme de base

Un modeste programme d'impression d'un fichier properties du premier paragraphe de ce chapitre pourrait donc finalement s'écrire :

Impression.java

```
/*  
 * Impression.java  
 * Created on 18 mai 2005, 21:32  
 */  
/**  
 * @author Vilvens  
 */
```

```
import java.io.*;
import javax.print.*;

public class Impression
{
    public static void main(String[] args)
    {
        try
        {
            FileInputStream fis = new FileInputStream
                ("c:\\java-sun-
application\\Internationalisation\\TextesAeroport_en_UK.properties");
            System.out.println("Fichier ouvert");

DocFlavor df = DocFlavor.INPUT_STREAM.AUTOSENSE;
System.out.println("DocFlavor instancié");

PrintService service = PrintServiceLookup.lookupDefaultPrintService();
System.out.println("Service trouvé");

DocPrintJob dpj = service.createPrintJob();
System.out.println("Job créé");

Doc doc = new SimpleDoc(fis, df, null);
System.out.println("Document créé");

dpj.print(doc, null);
System.out.println("Impression lancée");

fis.close();
}
catch (PrintException e)
{
    System.out.println("Erreur d'impression : " + e.getMessage());
}
catch (IOException e)
{
    System.out.println("Erreur d'IO : " + e.getMessage());
}
}
```

4. Les expressions régulières

4.1 Le scalpel des chaînes de caractères

En abrégé, on pourrait dire que les expressions régulières [*Regular expression*] sont un outil permettant d'établir si une chaîne de caractères répond ou pas à un modèle donné (définis par un groupe de séparateurs, ceux-ci étant formés de un ou de plusieurs caractères) et de découper cette chaîne de caractères en les éléments constitutifs implicitement définis par ce modèle.

En ce sens, cet outil fait donc penser au StringTokenizer déjà présenté dans le chapitre VI, mais en beaucoup plus puissant. Popularisée par des langages comme PERL et PHP, ces expressions régulières sont utilisées dans des domaines très divers. Les deux cas les plus courants sont :

- ◆ les vérifications d'URLs, d'adresses e-mail, d'entrées alphanumériques à structure précise (comme un numéro de téléphone ou une date);
- ◆ les extractions d'éléments de fichiers de configurations ou fichiers de directives,

Il existe une norme POSIX pour les expressions régulières; mais elle n'est pas la seule – ainsi, PERL a donné une norme PCRE (Perl Compatible Regular Expression).

4.2 Les motifs

Pour définir le groupe de caractères recherchés, il faut définir un **motif** [*pattern*]. Par exemple, le motif "[**a-z**]" désigne un caractère quelconque appartenant à la plage des caractères allant de 'a' à 'z'. Ou encore, "**a{2,4}**" désigne une chaîne qui peut être "aa", "aaa" ou "aaaa".

Plus précisément, pour former un motif, on utilise, outre les littéraux que sont les lettres ou les chiffres :

- ◆ les symboles de début et fin de chaîne :

^ : désigne le début de chaîne recherché; par exemple, "^ISO" désigne une ligne qui commence par "ISO";

\$: désigne la fin de chaîne recherché; par exemple, "ISO\$" désigne une ligne qui se termine par "ISO";

- ◆ le point : il désigne n'importe quel caractère;

◆ les symboles quantificateurs : ils s'utilisent comme suffixes du caractère ou de la chaîne considérée pour indiquer le nombre de répétitions :

* : 0 ou plusieurs occurrences; par exemple, "a*" correspond à "a", "aa", "aaa", ... ou pas de "a" du tout;

+ : 1 ou plusieurs occurrences; par exemple, "a+" correspond à "a", "aa", "aaa", ... - donc au moins un "a";

? : 0 ou 1 occurrence; par exemple, "a?" correspond à "a" ou pas de "a" du tout.

◆ les intervalles de reconnaissance : utilisés en suffixe d'un littéral, ils s'expriment au moyen des accolades {} et fixent le nombre de répétitions :

- un seul chiffre donne le nombre exact de caractères; par exemple, "a{2}" correspond exactement à "aa";

- un seul chiffre suivi d'une virgule donne le nombre minimum de caractères; par exemple, "a{2,}" correspond à un minimum de deux 'a' consécutifs, donc "aa", "aaa", "aaaaa",

- deux chiffres séparés par une virgule donnent le nombre minimum et maximum de caractères; par exemple, "a{2,4}" correspond uniquement à "aa", "aaa" et "aaaa"

- ◆ les classes de caractères : il s'agit d'énumérer, éventuellement avec des abréviations, les caractères à reconnaître; ces classes s'expriment avec des crochets [] ; on peut rencontrer :
 - [<liste de caractères>] : par exemple, "gra[dv]e" désigne en fait "grade" ou "grave";
 - [<caractère>-<caractère>] : désigne toute une plage; par exemple, "gr[1-6]" désigne "gr1", "gr2", ..., "gr6";
 - [^<liste de caractères>] : désigne la classe "complémentaire", donc regroupe tous les caractères qui ne sont pas ceux cités; par exemple, "[^abcd]" désigne tout caractère qui n'est pas "a", "b", "c" ou "d";
- il existe des classes prédéfinies comme
 - \p{Alpha} (POSIX) ([:alpha:] en Perl) : n'importe quelle lettre;
 - \p{Digit} (POSIX) ([:digit:] en Perl) : (n'importe quel chiffre);
 - \p{Lower} et \p{Upper} (POSIX) ([:lower:] et [:upper:] en Perl) : n'importe quelle lettre en minuscule et majuscule);
 - \p{Cntrl} (POSIX) ([:cntrl:] en Perl) : caractères d'échappement;etc.
- ◆ l'alternative : symbolisée par "|", elle permet de permettre deux possibilités; par exemple, "(Denys|Pierre)_Vilvens" correspond à "Denys_Vilvens" ou "Pierre_Vilvens".

Comme d'habitude, les parenthèses servent à modifier la portée d'un opérateur et le symbole "\\" permet de considérer le symbole suivant comme un caractère spécial comme en fait normal (sauf '-' et ']').

Le sujet des expressions régulières est fort vaste : des livres entiers sont consacrés au sujet ! Limitons-nous ici à quelques manipulations élémentaires ...

4.3 Les expressions régulières en Java

Le JDK fournit un package `java.util.regex` qui ne comporte que deux classes, mais qui est bien suffisant pour faire fonctionner le mécanisme des expressions régulières.

a) La classe **Pattern** a pour rôle de représenter une expression régulière. En fait, un tel objet est initialisé non pas au moyen d'un constructeur mais de la méthode de classe :

```
public static Pattern compile (String regex)
```

qui reçoit une chaîne de caractères exprimant une expression régulière et construit l'objet si aucune faute de syntaxe n'est détectée; dans le cas contraire, l'exception `PatternSyntaxException` est lancée (mais cette exception est "unchecked" – on peut donc l'ignorer sans que le compilateur ne proteste).

b) La classe **Matcher** a pour tâche d'analyser une chaîne de caractères en fonction d'une expression régulière donnée par un objet `Pattern`. Un tel objet `Matcher` s'obtient au moyen de la méthode de `Pattern` :

```
public Matcher matcher(CharSequence input)
```

`CharSequence` n'est qu'un interface apparu dans le JDK 1.4 et qui est implémenté notamment par `String`. Donc, la méthode considérée reçoit bien la chaîne à analyser. L'objet `Matcher` ainsi obtenu possède toute une série de méthodes d'examen, dont nous retiendrons :

public boolean **find()**

Cherche la sous-chaîne suivante en concordance avec le pattern. Classiquement, on l'utilise itérativement pour parcourir toute la chaîne à examiner. La sous-chaîne valide en questions peut s'obtenir au moyen de la méthode :

public String **group()**

Fournit la sous-chaîne en question

public boolean **matches()**

Vérifie si l'entièreté de la chaîne est en accord avec les règles formulées par l'expression régulière.

public boolean **lookingAt()**

Vérifie si il est possible de trouver dans la chaîne une sous-chaîne en accord avec les règles formulées par l'expression régulière.

4.4 Un exemple de recherche et de test

Le modeste programme suivant illustre l'utilisation de ces deux classes et de leurs méthodes :

- on recherche dans une chaîne toutes les plaques de voitures ayant le format <3 lettres – 3 chiffres>;
- on teste si une plaque est correctement formée;
- on balaie un fichier de noms de clients pour vérifier si chaque nom est construit selon la règle "une majuscule suivie d'une séquence de minuscules et/ou d'espaces".

PlaqueVoiture.java

```
/*
 * PlaqueVoiture.java
 * Created on 27 mai 2005, 17:59
 */
/**
 * @author Vilvens
 */
import java.util.regex.*;
import java.util.*;
import java.io.*;

public class PlaqueVoiture
{
    private static String REP_TRAVAIL = "c:\\java-sun-application\\ExpressionReguliere\\";

    public static void main(String[] args)
    {
        // -- Recherches de plaques
        String plaqueVoiturePattern = "[A-Z][A-Z][A-Z][0-9][0-9][0-9]";
        Pattern p = Pattern.compile(plaqueVoiturePattern);

        String essai = "41derf12AVF459rf125tgbh515ADED741";
        StringBuffer carIgn = new StringBuffer();

        Matcher m = p.matcher(essai);
```

```

while (m.find())
{
    String gr = m.group();
    System.out.println("plaque trouvée = " + gr);
    m.appendReplacement(carIgn,"");
}
System.out.println("Caractères ignorés : " + carIgn);

// -- Test d'une plaque
String plaque = "ETJ988";
Matcher mp = p.matcher(plaque);
if(mp.matches())
    System.out.println("Plaque valide");
else
    System.out.println("Plaque NON valide");

// -- Test d'une série de noms
String nomPattern = "[A-Z]( [a-z] | [èéàùê])*"; Pattern pn = null;
try
{
    pn = Pattern.compile(nomPattern);
}
catch (PatternSyntaxException e) // Unchecked exception
{
    System.out.println("Erreur exp. reg : " + e.getMessage());
}

try
{
    System.out.print("Nom du fichier à étudier : ");
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    String nomFichier = in.readLine();
    FileReader fr = new FileReader (REP_TRAVAIL + nomFichier);
    BufferedReader br = new BufferedReader(fr);
    String nom;
    while ((nom=br.readLine()) != null)
    {
        System.out.print("** Nom testé : " + nom + " --> ");
        Matcher mn = pn.matcher(nom);

        if (mn.find())
        {
            String gr = mn.group();
            System.out.print("- Nom trouvé = " + gr + " - ");
        }

        if(mn.matches())
            System.out.print("match : Nom valide / ");
        else
            System.out.print("match : Nom NON valide / ");
}

```

```
if(mn.lookingAt())
    System.out.println("lookingAt : Nom valide");
else System.out.println("lookingAt : Nom NON valide");
}
}
catch (FileNotFoundException e)
{
    System.out.println("Exception FNF flux : " + e.getMessage());
}
catch (IOException e)
{
    System.out.println("Exception IO flux : " + e.getMessage());
}
}
}
```

En exécutant ce programme avec un fichier de données du type suivant (le nom "Clermont " se termine par un espace) :

fichier NomsClients.txt

```
Vilvens
Charlet
De fooz
KUty
Clermont
R2D2
R deux d deux
de la tronche en bière
De la tronche en biere
```

on obtient :

```
plaque trouvée = AVF459
plaque trouvée = DED741
Caractères ignorés : 41derf12rf125tgbh515A
Plaque valide
Nom du fichier à étudier : NomsClients.txt
** Nom testé : Vilvens --> - Nom trouvé = Vilvens - match : Nom valide / lookingAt : Nom valide
** Nom testé : Charlet --> - Nom trouvé = Charlet - match : Nom NON valide / lookingAt : Nom NON valide
** Nom testé : De fooz --> - Nom trouvé = De fooz - match : Nom valide / lookingAt : Nom valide
** Nom testé : KUty --> - Nom trouvé = K - match : Nom NON valide / lookingAt : Nom valide
** Nom testé : Clermont --> - Nom trouvé = Clermont - match : Nom valide / lookingAt : Nom valide
** Nom testé : R2D2 --> - Nom trouvé = R - match : Nom NON valide / lookingAt : Nom valide
```

```
** Nom testé : R deux d deux --> - Nom trouvé = R deux d deux - match : Nom valide /  
lookingAt : Nom valide  
** Nom testé : de la tronche en bière --> match : Nom NON valide / lookingAt : Nom NON  
valide  
** Nom testé : De la tronche en biere --> - Nom trouvé = De la tronche en biere - match :  
Nom valide / lookingAt : Nom valide
```

Si l'on souhaite admettre les caractères accentués francophones, il suffit de compléter le motif de ceux-ci :

```
String nomPattern = "[A-Z]([a-z]|([èéàùê])*";
```

Un nom comme "De la tronche en bière" sera alors considéré comme valide.

5. Les compressions de fichiers

5.1 Histoires de zip

Il existe un certain nombre d'algorithmes de compression de fichiers. Cependant, les formats gzip et surtout zip sont les plus populaires.

La compression gzip utilise un codage connu sous le nom de Lempel-Ziv (LZ77). Elle est fort utilisée sous UNIX, où le fichier résultant de la compression d'un fichier donné a l'extension .gz. Comme cette méthode de compression ne s'applique qu'à un fichier à la fois, il est d'usage de regrouper plusieurs fichiers comprimés dans un fichier .tar, ce qui donne alors un fichier global d'extension .tar.gz. La taux de compression [*ratio*] se calcule bien logiquement selon :

$$\text{ratio} = \frac{(\text{taille_sans_compression} - \text{taille_après_compression}) * 100}{\text{taille_sans_compression}}$$

On estime que le taux de compression est de l'ordre de 60 à 70% pour des fichiers de type code source en anglais, ce qui est excellent par rapport à d'autres algorithmes (comme celui de Huffman).

La compression zip permet de compresser plusieurs fichiers en un seul fichier résultant, le fichier archive. En fait, un tel fichier zip est analogue à un fichier jar, si ce n'est que celui-ci possède en plus un répertoire META-INF avec au minimum un fichier manifeste à l'intérieur.

Tous les outils concernant gzip et zip se trouvent dans le package `java.util.zip`. Nous allons nous concentrer sur la technique la plus courante, soit celle du zip.

5.2 Les sommes de contrôle

Le calcul de deux sommes de contrôle (*checksum*), l'une calculée à la lecture du fichier origine et l'autre lors de l'écriture, permet de s'assurer raisonnablement qu'il n'y a pas eu de pertes de données. Le CRC est bien connu dans le domaine des checksum, mais est moins fiable et moins rapide que l'algorithme d'Adler. Ces deux algorithmes correspondent aux deux classes **CRC32** et **Adler32**. Toutes deux implémentent l'interface **Checksum** dont les deux méthodes importantes sont :

public long getValue()

Pour obtenir la valeur de la somme de contrôle

public void reset()

Pour réinitialiser cette somme avant un nouveau calcul.

Bien logiquement, une telle somme de contrôle se calcule en lisant un fichier : un flux particulier sera bien logiquement chargé de ce travail

- ◆ lors de la lecture, un flux instance de **CheckedInputStream** gère une somme de contrôle;
- ◆ lors de l'écriture, un flux instance de **CheckedOutputStream** gère une autre somme de contrôle.

Ces deux flux sont bien entendu des classes dérivées de **FilterInputStream** et **FilterOutputStream**. Ils sont construits sur des flux de plus bas niveau, avec des constructeurs réclamant la classe de calcul de la checksum :

public CheckedInputStream (InputStream in, Checksum cksum)
public CheckedOutputStream (OutputStream out, Checksum cksum)

Ils lisent ou écrivent leurs données sous formes de bytes au moyen respectivement de

public int read (byte[] buf, int off, int len) throws IOException

et

public void write (byte[] b, int off, int len) throws IOException

tandis qu'elles fournissent toutes deux la somme de contrôle au moyen de la méthode

public Checksum getChecksum()

La comparaison positive de ces deux sommes permettra de penser qu'il n'y a pas eu de problème lors du traitement.

5.3 Les entrées d'un fichier zip

Comme on pouvait s'y attendre, un fichier zip est référencé dans la programmation par une instance de la classe **ZipFile**. "Référencé", pas "représenté" : en effet, un tel objet ne contient pas les éléments constituants, mais est en fait un container d'entrées donnant accès chacune à un fichier du zip. C'est ce que permet sa méthode :

public Enumeration entries()

Une telle entrée est en fait représentée par un objet instance de **ZipEntry**, dont le constructeur est tout simplement

public ZipEntry (String name)

Ses méthodes permettent d'obtenir toutes les informations sur cette entrée, comme par exemple :

```
public String getName()  
public long getSize()  
public long getCompressedSize()  
public long getCrc()
```

etc.

La classe ZipFile possède une méthode

```
public InputStream getInputStream (ZipEntry entry) throws IOException
```

dont le rôle est de fournir un flux du l'entrée précisée.

5.4 La création d'un fichier zip

Concrètement, si l'on souhaite créer un fichier zip, on instanciera un flux **ZipOutputStream**, dérivé de FilterOutputStream par l'intermédiaire d'une classe DeflaterOutputStream qui sert aussi de mère pur la classe sœur gérant les gzip. Le constructeur est classiquement :

```
public ZipOutputStream (OutputStream out)
```

- autrement dit demande le flux fichier dans lequel il lui faudra écrire, soit par exemple :

```
zos = new ZipOutputStream( new BufferedOutputStream(  
    new FileOutputStream("C:\\java-sun-application\\Internationalisation\\properties.zip")));
```

On peut ensuite paramétrier la compression que ce flux va réaliser sur ses entrées au moyen des méthodes :

```
public void setMethod(int method)
```

Elle détermine si le fichier traité doit être effectivement compressé (indiqué par la constante DEFLATED) ou simplement engrangé tel quel (indiqué par la constante STORED).

```
public void setLevel(int level)
```

Dans le cas où la méthode fixée ci-dessus, on peut définir le niveau de compression qui va de 0 à 9. Les constantes suivantes, définies dans la classe Deflater, peuvent être utilisées :

public static final int NO_COMPRESSION	// niveau = 0
public static final int BEST_SPEED	// niveau = 1
public static final int BEST_COMPRESSION	// niveau = 9
public static final int DEFAULT_COMPRESSION	// niveau = 5

Dans notre cas, cela pourrait être :

```
zos.setMethod(ZipOutputStream.DEFLATED);  
zos.setLevel(Deflater.BEST_COMPRESSION);
```

On peut à présent remplir le zip en créant un objet ZipEntry par fichier à intégrer et en l'enregistrant au moyen de la méthode ZipOutputStream :

```
public void putNextEntry (ZipEntry e) throws IOException
```

Il ne reste plus qu'à

- ◆ lire le fichier à intégrer au moyen d'un CheckedInputStream construit sur un FileInputStream associé au fichier;
- ◆ écrire dans le fichier ZipOutputStream en passant par un CheckedOutputStream.

La comparaison des deux checksums obtenus permettra de savoir si tout semble s'être bien passé.

5.5 Un exemple console de création de fichier zip

Le programme suivant illustre nos propos. Il va créer un fichier zip ("properties.zip") qui contiendra les fichiers properties évoqués dans le paragraphe consacré à l'internationalisation (soit TextesAeroport_en_UK.properties et TextesAeroport_fr_FR.properties). Si le répertoire source est codé à la dure, nous allons par contre demander à notre application d'y retrouver elle-même les fichiers d'extension ".properties". Pour sélectionner les fichiers qui nous intéressent, la classe File propose, entre autres, une méthode :

```
public String[] list (FilenameFilter filter)
```

Celle-ci fournit la liste des fichiers se trouvant dans le répertoire correspondant à l'objet File qui appelle la méthode et qui satisfont à un critère de sélection défini dans le paramètre de la méthode. Ce paramètre est un objet qui implémente l'interface **FilenameFilter**, dont la seule méthode déclarée est :

```
public boolean accept (File dir, String name)
```

Le rôle de cette méthode est de retourner true ou false, selon que le fichier dont le nom est le 2^{ème} argument et se trouvant effectivement dans le répertoire désigné par le 1^{er} argument doit être retenu ou pas. Dans notre cas, ce sont les seuls fichiers d'une extension donnée (en l'occurrence "properties") qui doivent être retenus, si bien que nous pouvons construire le modeste filtreur suivant, dont le constructeur reçoit l'extension cherchée :

Filtre.java

```
/*
 * Filtre.java
 * Created on 1 juin 2005, 12:07
 */

/**
 * @author Vilvens
 */
import java.io.*;
import java.util.*;
```

```
public class Filtre implements FilenameFilter
{
    private String extension;

    public Filtre (String e)
    {
        extension = e;
    }

    public boolean accept (File f, String s)
    {
        StringTokenizer parser = new StringTokenizer(s, ".");
        String tok = null;
        while (parser.hasMoreTokens()) tok = parser.nextToken();
        if (extension.equals(tok)) return true;
        else return false;
    }
}
```

Avec un tel filtre, on obtiendra les noms des fichiers recherchés par :

```
File fi = new File("C:\\java-sun-application\\Internationalisation\\");
Filtre f = new Filtre("properties");
String[] listeNomsFichiers = fi.list(f);
```

Donc, finalement :

Zip.java
<pre>/* * Zip.java * Created on 1 juin 2005, 11:50 */ /** * @author Vilvens */ import java.io.*; import java.util.zip.*; public class Zip { public static void main(String[] args) { File <i>fi</i> = new File("C:\\java-sun-application\\Internationalisation\\"); Filtre <i>f</i> = new Filtre("properties"); String[] listeNomsFichiers = <i>fi.list</i>(<i>f</i>); for (int i=0; i<listeNomsFichiers.length; i++) System.out.println(i + " . " + listeNomsFichiers[i]); String rep = "C:\\java-sun-application\\Internationalisation\\"; } }</pre>

```

ZipOutputStream zos = null;
try
{
    zos = new ZipOutputStream (new BufferedOutputStream (new FileOutputStream
        ("C:\\java-sun-application\\Internationalisation\\properties.zip")));
    zos.setMethod (ZipOutputStream.DEFLATED);
    zos.setLevel (Deflater.BEST_COMPRESSION);

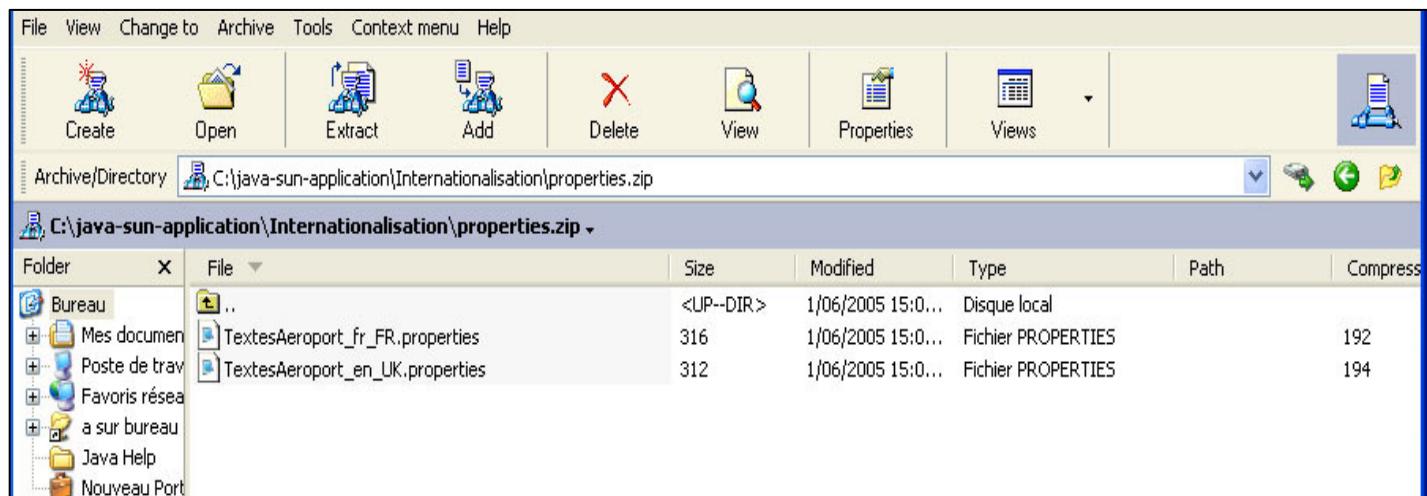
    CheckedOutputStream cos = new CheckedOutputStream(zos, new Adler32());
    byte[] buf = new byte [8192];

    for (int i=0; i<listeNomsFichiers.length; i++)
    {
        ZipEntry ze = new ZipEntry (listeNomsFichiers[i]);
        zos.putNextEntry(ze);
        CheckedInputStream cis = new CheckedInputStream(new BufferedInputStream
            (new FileInputStream(rep+listeNomsFichiers[i])), new Adler32());
        int n=0;
        while ( (n=cis.read(buf))>0)
            cos.write(buf, 0, n);

        if (cos.getChecksum().getValue() != cis.getChecksum().getValue())
            System.out.println("Mauvais checksum !");
        else
            zos.closeEntry();
        cos.getChecksum().reset();
    }
    cos.close();
}
catch (IOException e) { System.out.println(e.getMessage()); }
}
}

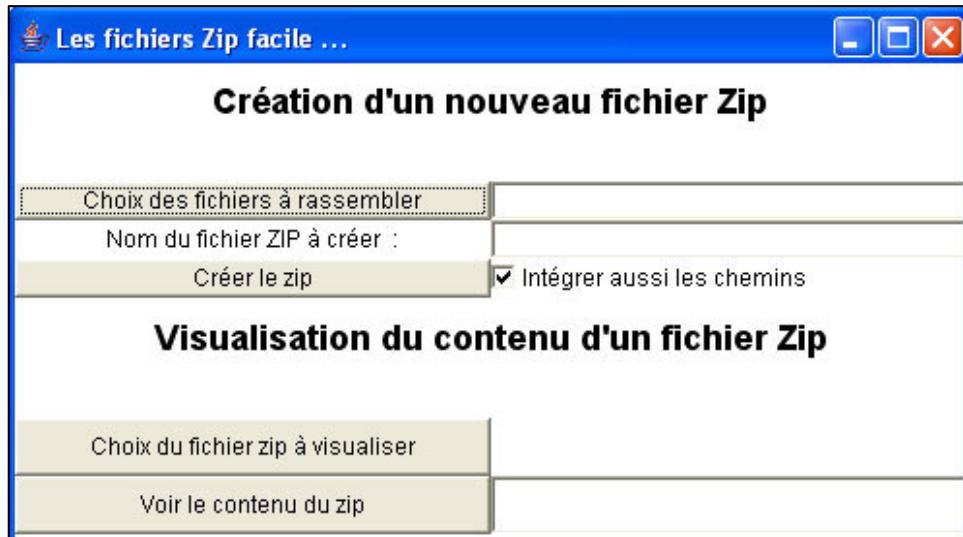
```

On peut vérifier que le fichier zip obtenu contient bien les fichiers prévus :



5.6 Un GUI pour créer et visualiser les zips

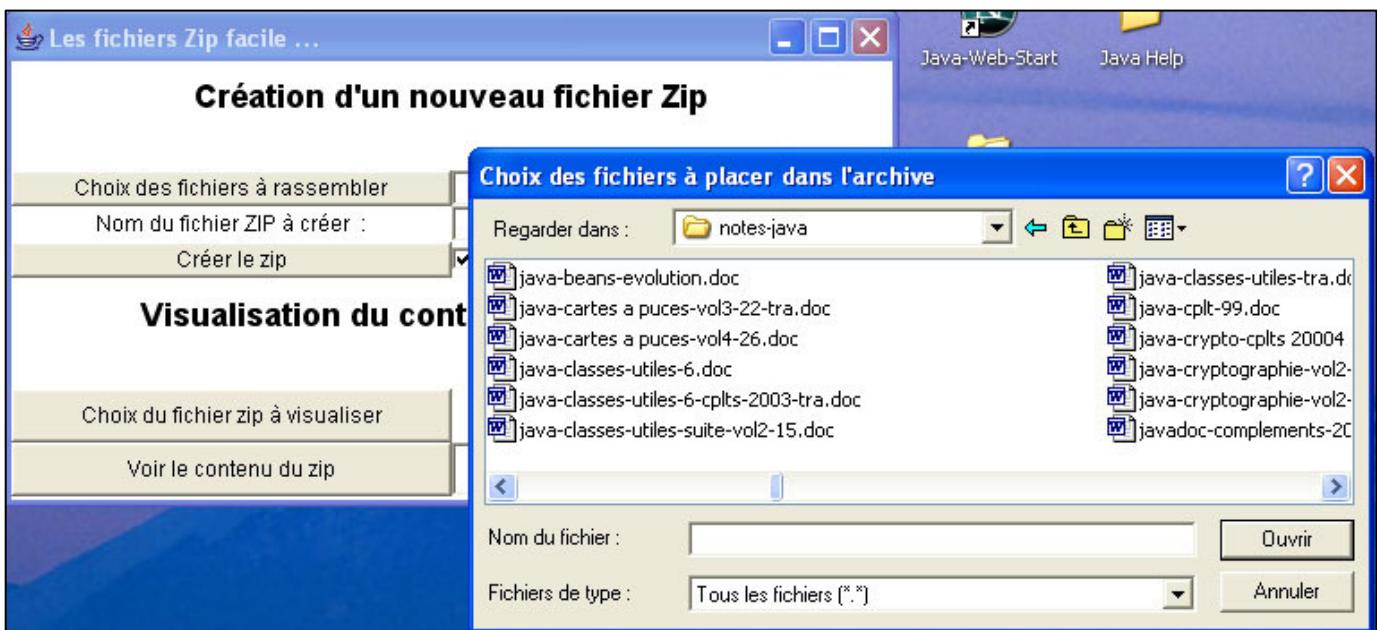
Dans la foulée de cette application console, nous allons encore, pour terminer ce survol des techniques de compression, nous attaquer à la confection de la petite application GUI suivante :



Les fonctionnalités sont bien claires.

a) La création d'un fichier Zip

L'appui sur le bouton "Choix des fichiers à rassembler" fera apparaître une boîte de dialogue de choix de fichier :



Cette boîte appartient en fait à la librairie Java : on trouve en effet dans le package javav.awt une classe **FileDialog** qui prend en charge les fonctionnalités de choix d'un fichier. Son constructeur le plus complet est :

```
public FileDialog (Frame parent, String title, int mode)
```

où le 3^{ème} paramètre peut valoir l'une des deux valeurs définies comme constantes dans la classe :

```
public static final int LOAD
public static final int SAVE
```

selon que l'on recherche un fichier à lire ou qu'il s'agit plutôt de trouver un endroit où écrire un fichier. Lorsque cette boîte est refermée, on peut lui demander le nom du fichier choisi ainsi que son répertoire au moyen de ses méthodes :

```
public String getFile()
public String getDirectory()
```

Les autres opérations du création du fichier zip sont analogues à celles de l'exemple précédent, si ce n'est que l'on offre à l'utilisateur de créer un fichier zip avec les chemins originaux ou pas (dans ce dernier cas, on "remet tout à plat"). Il faut donc conserver simultanément

- ◆ le nom complet de chaque fichier sélectionné (la boîte de liste du GUI s'en chargera);
- ◆ les noms "abrégés" de chacun de ces fichiers – une LinkedList nommée listeNomsFichiersAbreges s'en chargera.

La première version de notre programme s'écrira donc (passons avec dédain sur les histoires de Panel) :

FenZipGui.java (1)

```
/*
 * FenZipGui.java
 * Created on 1 juin 2005, 18:25
 */
/**
 * @author Vilvens
 */
import java.awt.*;
import java.io.*;
import java.util.*;
import java.util.zip.*;

public class FenZipGui extends java.awt.Frame
{
    private String dernierRep;
    private LinkedList listeNomsFichiersAbreges;
    private String nomFichierZip;

    public FenZipGui()
    {
        initComponents();
        dernierRep = ". "; listeNomsFichiersAbreges = new LinkedList();
    }
}
```

```

private void initComponents() { ... }

private void BChoixFichierActionPerformed(java.awt.event.ActionEvent evt)
{
    FileDialog fd = new FileDialog(this, "Choix des fichiers à placer dans l'archive",
                                    FileDialog.LOAD);
    fd.setVisible(true);
    String rep = fd.getDirectory();
    String fichier = fd.getFile();
    System.out.println("Fichier choisi : " + fichier);
    System.out.println("Répertorie choisi : " + rep);
    ListeFichiers.add(rep + fichier);
    listeNomsFichiersAbreges.add(fichier);
    dernierRep = rep;
}

private void BCrerZipActionPerformed(java.awt.event.ActionEvent evt)
{
    String nomFichierZip = ZENomFichierZipACreer.getText();
    String[] listeNomsFichiers = ListeFichiers.getItems();
    System.out.println("Liste des noms complets de fichiers");
    for (int i=0; i<listeNomsFichiers.length; i++)
        System.out.println(i + " . " + listeNomsFichiers[i]);
    System.out.println("Liste des noms abrégés de fichiers");
    for (int i=0; i<listeNomsFichiersAbreges.size(); i++)
        System.out.println(i + " . " + (String)listeNomsFichiersAbreges.get(i));

ZipOutputStream zos = null;
try
{
    zos = new ZipOutputStream(new BufferedOutputStream
        (new FileOutputStream(dernierRep + nomFichierZip)));
    zos.setMethod(ZipOutputStream.DEFLATED);
    zos.setLevel(Deflater.BEST_COMPRESSION);

    CheckedOutputStream cos = new CheckedOutputStream(zos, new Adler32());
    byte[] buf = new byte [8192];

    for (int i=0; i<listeNomsFichiers.length; i++)
    {
        ZipEntry ze = null;
        if (CBAvecChemins.getState())
            ze = new ZipEntry(listeNomsFichiers[i]);
        else
            ze = new ZipEntry((String)listeNomsFichiersAbreges.get(i));
        zos.putNextEntry(ze);
        CheckedInputStream cis = new CheckedInputStream(new BufferedInputStream
            (new FileInputStream(listeNomsFichiers[i])), new Adler32());

        int n=0;
    }
}

```

```

while ( (n=cis.read(buf))>0)
    cos.write(buf, 0, n);

if (cos.getChecksum().getValue() != cis.getChecksum().getValue())
    System.out.println("Mauvais checksum !");
else
    zos.closeEntry();
    cos.getChecksum().reset();
}
cos.close();
}
catch (IOException e) { System.out.println(e.getMessage()); }
}

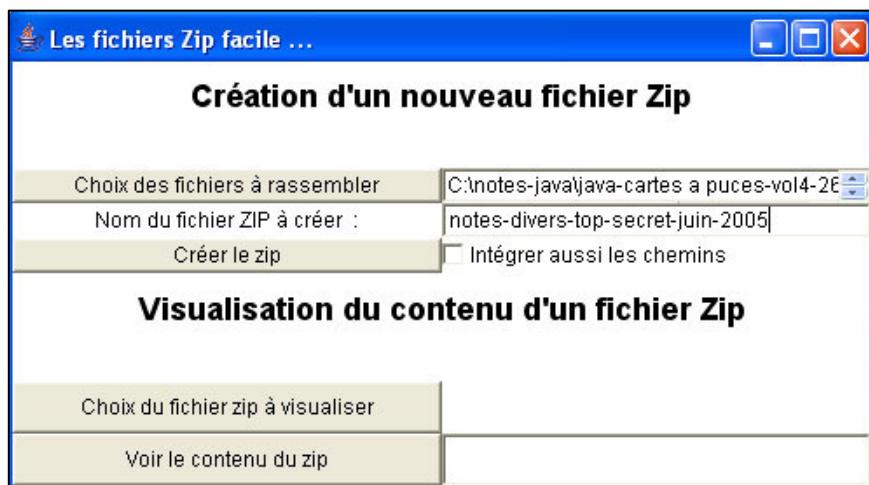
private void exitForm(java.awt.event.WindowEvent evt) { System.exit(0); }

public static void main(String args[])
{
    new FenZipGui().show();
}

private java.awt.Panel panel1;
private java.awt.Label label2;
private java.awt.Panel PanelVisualisation;
private java.awt.Panel PanelCreation;
private java.awt.TextField ZENomFichierZipACreer;
private java.awt.Button BVoirZip;
private java.awt.List ListeFichiers;
private java.awt.Button BChoixFichier;
private java.awt.Panel PanelTitreCreer;
private java.awt.Checkbox CBAvecChemins;
private java.awt.Button BCrerZip;
private java.awt.Label label1;
}
}

```

Une petite démonstration :



avec effet sur la console :

```
Fichier choisi : java-classes-utiles-6.doc
Répertorie choisi : C:\notes-java\
Fichier choisi : java-cryptographie-vol2-13.doc
Répertorie choisi : C:\notes-java\
Fichier choisi : CryptageSymetrique.java
Répertorie choisi : C:\java-sun-application\Crypto\
Fichier choisi : ClesPourCryptageAsymétrique.java
Répertorie choisi : C:\java-sun-application\Crypto\
Fichier choisi : Utilisateurs.properties
Répertorie choisi : C:\java-sun-application\Hopital\
Fichier choisi : java-cartes a puces-vol4-26.doc
Répertorie choisi : C:\notes-java\
```

Liste des noms complets de fichiers

- 0 . C:\notes-java\java-classes-utiles-6.doc
- 1 . C:\notes-java\java-cryptographie-vol2-13.doc
- 2 . C:\java-sun-application\Crypto\CryptageSymetrique.java
- 3 . C:\java-sun-application\Crypto\ClesPourCryptageAsymétrique.java
- 4 . C:\java-sun-application\Hopital\Utilisateurs.properties
- 5 . C:\notes-java\java-cartes a puces-vol4-26.doc

Liste des noms abrégés de fichiers

- 0 . java-classes-utiles-6.doc
- 1 . java-cryptographie-vol2-13.doc
- 2 . CryptageSymetrique.java
- 3 . ClesPourCryptageAsymétrique.java
- 4 . Utilisateurs.properties
- 5 . java-cartes a puces-vol4-26.doc

b) La visualisation d'un fichier Zip

Passons à présent à la visualisation d'un fichier zip préalablement sélectionné au moyen d'une `FileDialog`. Nous avions bien pensé pour cette dernière à définir un filtre afin que seuls les fichiers `*.zip` apparaissent, mais l'aide du JDK 1.4 nous a gentiment signalé que cela ne fonctionnait pas avec Windows ;-) ...

La seule véritable nouveauté est ici l'utilisation de la méthode `entries()` de la classe `ZipFile`, déjà signalée. On parcourt l'énumération obtenue : on obtient ainsi à chaque étape un objet `ZipEntry` dont nous pouvons extraire tous les renseignements utiles. Les compléments à apporter à notre application sont donc :

FenZipGui.java (2)

```
/*
 * FenZipGui.java
 * Created on 1 juin 2005, 18:25
 */
/**
 * @author Vilvens
 */
import java.awt.*;
```

```

import java.io.*;
import java.util.*;
import java.util.zip.*;

public class FenZipGui extends java.awt.Frame
{
    ...
    private void BChoixFichiersAVisualiserActionPerformed
        (java.awt.event.ActionEvent evt)
    {
        FileDialog fd = new FileDialog(this, "Choix du fichier à visualiser",
            FileDialog.LOAD);
        fd.setVisible(true);
        String rep = fd.getDirectory();
        String fichier = fd.getFile();
        System.out.println("Fichier choisi : " + fichier);
        System.out.println("Répertoire choisi : " + rep);
        nomFichierAVisualiser = rep+fichier;
        ZTNomFichierZipAVisualiser.setText(nomFichierAVisualiser);
    }

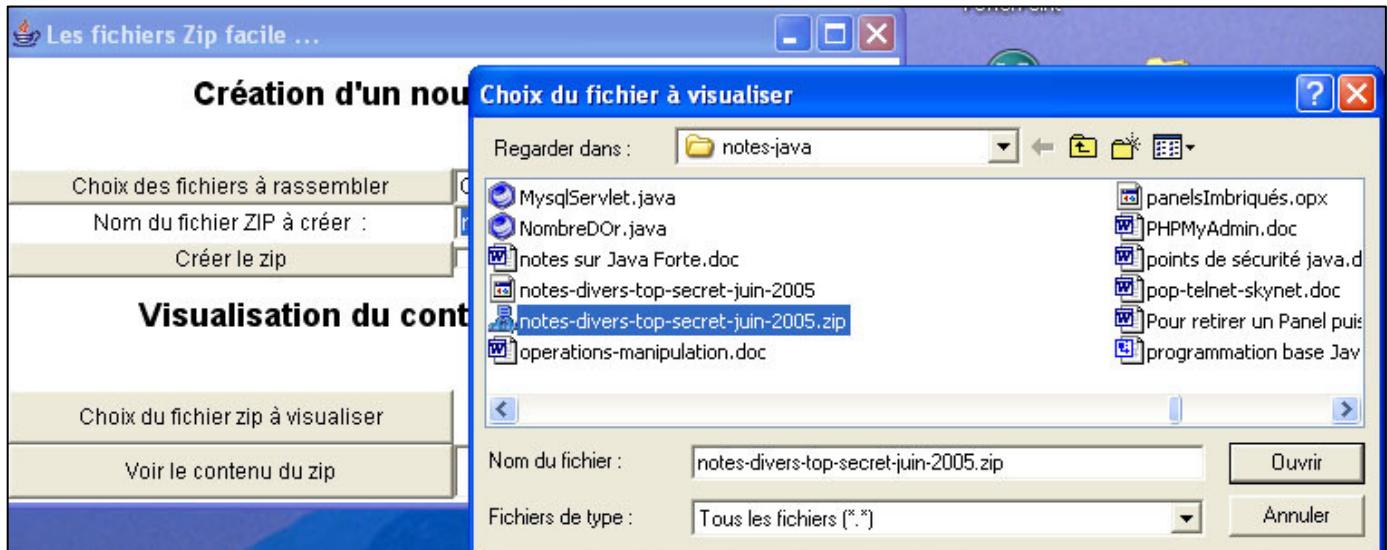
    private void BVoirZipActionPerformed(java.awt.event.ActionEvent evt)
    {
        File f = new File (nomFichierAVisualiser);
        ZipFile zf = null;
        try
        {
            zf = new ZipFile(f);
        }
        catch (ZipException e)
        {
            System.out.println("Erreur zip : " + e.getMessage());
        }
        catch (IOException e)
        {
            System.out.println("Erreur E/S : " + e.getMessage());
        }
        Enumeration e = zf.entries();
        while (e.hasMoreElements())
        {
            ZipEntry ze = (ZipEntry)e.nextElement();
            String infos = ze.getName() + " : " + ze.getSize() + " - " + ze.getCompressedSize();
            System.out.println(infos);
            ListeFichiersZip.add(infos);
        }
    }

    ...
    private java.awt.Label ZTNomFichierZipAVisualiser;
    private java.awt.Button BChoixFichiersAVisualiser;
}

```

```
private java.awt.Panel PanelVisualisation;  
private java.awt.Button BVoirZip;  
private java.awt.List ListeFichiersZip;  
}
```

La suite de la démonstration :



Le résultat dans le GUI :



et sur la console :

```
Fichier choisi : notes-divers-top-secret-juin-2005.zip  
Répertorie choisi : C:\notes-java\  
java-classes-utiles-6.doc : 1992704 - 1077325  
java-cryptographie-vol2-13.doc : 1767424 - 1002244  
CryptageSymetrique.java : 1579 - 628  
ClesPourCryptageAsymétrique.java : 1588 - 605  
Utilisateurs.properties : 135 - 112  
java-cartes a puces-vol4-26.doc : 2132480 - 1453488
```

6. JNI : les méthodes natives

6.1 Aux frontières de la portabilité

Il peut arriver qu'une application Java, portable en utilisant la machine virtuelle, ait cependant besoin d'utiliser des primitives système. Les exemples typiques de cette situation sont :

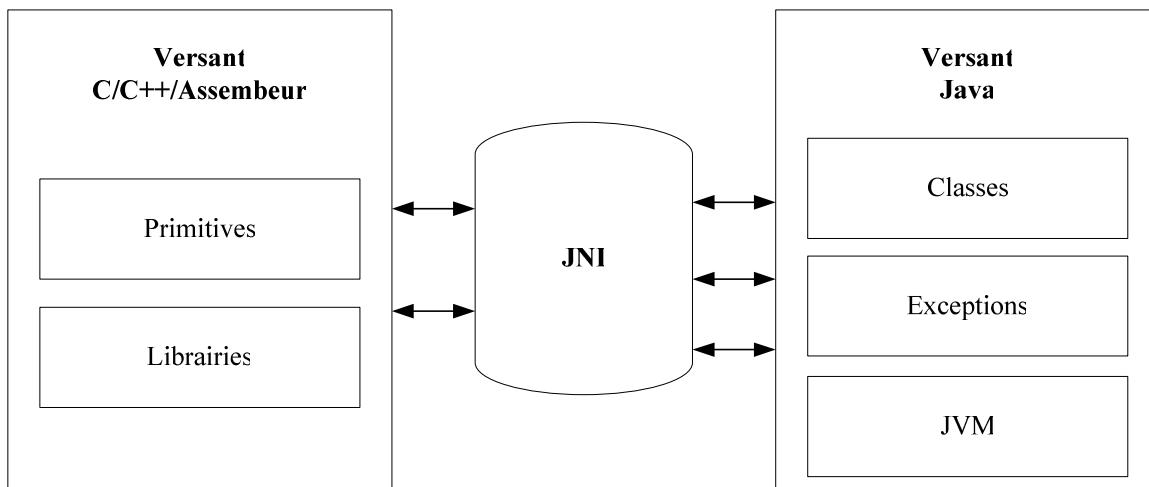
- ◆ l'obligation d'utiliser des appels du système d'exploitation (comme c'est le cas pour les threads par exemple : un thread Unix ne ressemble que de très loin à un thread Windows);
 - ◆ la nécessité d'utiliser une librairie existante dont la réécriture serait une perte de temps inacceptable;
 - ◆ la volonté d'utiliser des portions de code binaire optimisé dans un objectif de temps réel;
- autrement dit, quand on n'a pas le choix.

A priori, s'il en est ainsi, on peut oublier toute prétention de portabilité ... sauf si ces APIs présentent un interface constant pour des implémentations différentes selon les environnements (c'est la raison d'être de la norme POSIX) ! En effet, dans ce cas, le mécanisme du **Java Native Interface (JNI)** permet aux applications Java d'utiliser du code natif sur des machines virtuelles différentes, ce code natif binaire étant le résultat de compilations et éditions de liens de code source écrit en C, C++ ou Assembleur.

Plus précisément, JNI permet

- ◆ aux méthodes natives d'utiliser les objets Java;
- ◆ aux applications Java d'utiliser des méthodes natives;

et constitue donc l'interface entre le monde Java et le monde spécifique de la machine hôte. Le schéma traditionnel suivant résume ceci :



A nouveau, le sujet est vaste. Nous allons donc ici nous contenter de décrire ici un cas classique : une application Java veut utiliser une fonctionnalité disponible dans une librairie native.

6.2 L'application Java utilisatrice

Imaginons donc qu'une application Java souhaite afficher (et éventuellement utiliser) la valeur de variables d'environnement comme PROMPT ou PATH, lorsque ces variables existent sur la machine hôte. Bien sûr, l'objectif semble d'un intérêt relatif, mais ce modeste exemple sera suffisant pour exposer la technique à utiliser.

Le travail de retrouver les variables d'environnement et de les afficher sera donc confié à une méthode native, appelons-la "*afficheVariablesEnv()*", qui aura été définie comme une fonction écrite en C et placée dans une librairie (dll windows ou librairie partagée so Unix) nommée très finement "*JNIEnvironnementC*". Une classe Java va être chargée d'héberger cette méthode native, en plus d'une méthode "normale" :

ExploreEnvironnement.java

```
/*
 * ExploreEnvironnement.java
 */

package jnienvironment;

/**
 * @author Vilvens
 */

public class ExploreEnvironnement
{
    private String nomOS;

    public native void afficheVariablesEnv();

    public ExploreEnvironnement()
    {
        nomOS = System.getProperty("os.name");
    }

    public void afficheNomSystemeExploitation()
    {
        System.out.println("Système d'exploitation : " + nomOS);
    }
}
```

L'application Java utilisatrice de cette classe *ExploreEnvironnement* n'a pas grand' chose à faire :

- ♦ elle charge dynamiquement la librairie partagée nécessaire au moyen de la méthode de la classe *System* :

```
public static void loadLibrary(String libname)
```

- en fait, cet appel est équivalent à : *Runtime.getRuntime().loadLibrary(libname)*; l'exception **UnsatisfiedLinkError**, non nécessairement checkée, est susceptible d'être lancée si la librairie n'est pas trouvé (nous y reviendrons);

- ♦ appeler la fonction comme s'il s'agissait de l'une des méthodes de la classe ExploreEnvironnement :

UseExploreEnvironnement.java

```
/*
 * UseExploreEnvironnement.java
 */

package usejnienviromnement;
import jnienviromnement.*;

/**
 * @author Vilvens
 */

public class UseExploreEnvironnement
{
    static
    {
        try
        {
            System.loadLibrary("JNIEnvironnementC");
        }
        catch (UnsatisfiedLinkError e)
        {
            System.err.println("Echec au chargement de la librairie dynamique" + e);
            System.exit(1);
        }
    }

    public UseExploreEnvironnement() { }

    public static void main(String[] args)
    {
        ExploreEnvironnement ee = new ExploreEnvironnement();
        ee.afficheNomSystemeExploitation();
        ee.afficheVariablesEnv();
    }
}
```

En fait, l'appel de la méthode native peut également produire l'exception **UnsatisfiedLinkError** si la méthode n'est trouvée au link dynamique (il suffit pour cela que la signature version C et version java ne concorde pas).

On compile évidemment cette classe et cette application sous NetBeans pour obtenir des fichiers ExploreEnvironnement.class et UseExploreEnvironnement.class.

6.3 Le fichier header pour la méthode native

On peut s'étonner de mettre en place un décor avec l'appel d'une fonction que nous n'avons en fait pas encore écrite ! En fait, pour que le mécanisme fonctionne, la fonction C doit avoir un prototype bien précis. On peut demander à l'utilitaire **javah** (un membre de la famille java-javac-javadoc-jar-etc) de générer un fichier header où le prototype nécessaire à notre fonction aura été créé. Pour ce faire, il suffit de demander (sous Windows) :

```
C:\java-netbeans-application\JNIEnvironnement\build\classes>javah -jni
jnienvirnement.ExploreEnvironnement
```

On remarquera l'utilisation du nom complet de la classe. On peut constater que le fichier header a bien été créé :

```
C:\java-netbeans-application\JNIEnvironnement\build\classes>dir
Répertoire de C:\java-netbeans-application\JNIEnvironnement\build\classes

13/09/2007 08:11 <REP> .
13/09/2007 08:11 <REP> ..
13/09/2007 07:40 <REP> jnienvirnement
13/09/2007 08:11 324 jnienvirnement_ExploreEnvironnement.h
13/09/2007 07:40 <REP> usejnienvirnement
               1 fichier(s)      324 octets
               4 Rép(s) 121.671.581.696 octets libres
```

Editons ce fichier :

```
jnienvirnement_ExploreEnvironnement.h
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class jnienvirnement_ExploreEnvironnement */

#ifndef _Included_jnienvirnement_ExploreEnvironnement
#define _Included_jnienvirnement_ExploreEnvironnement
#ifndef __cplusplus
extern "C" {
#endif
/*
 * Class:  jnienvirnement_ExploreEnvironnement
 * Method: afficheVariablesEnv
 * Signature: ()V
 */
JNIEXPORT void JNICALL
Java_jnienvirnement_ExploreEnvironnement_afficheVariablesEnv
(JNIEnv *, jobject);

#ifndef __cplusplus
}
#endif
#endif
```

Evidemment, vu de loin, le nom donné à la fonction C semble assez lourd. En fait, il est construit selon la règle :

"**Java_**" + <nom complet de la classe> + " _ " + <nom de la méthode>

On remarquera aussi les deux paramètres, transparents en Java, qui désignent d'une part l'adresse de l'"environnement JNI" (c'est-à-dire la structure qui permet d'accéder aux véritables paramètres de la méthode Java) et d'autre part l'objet courant.

6.4 L'implémentation en C de la méthode native dans une librairie partagée

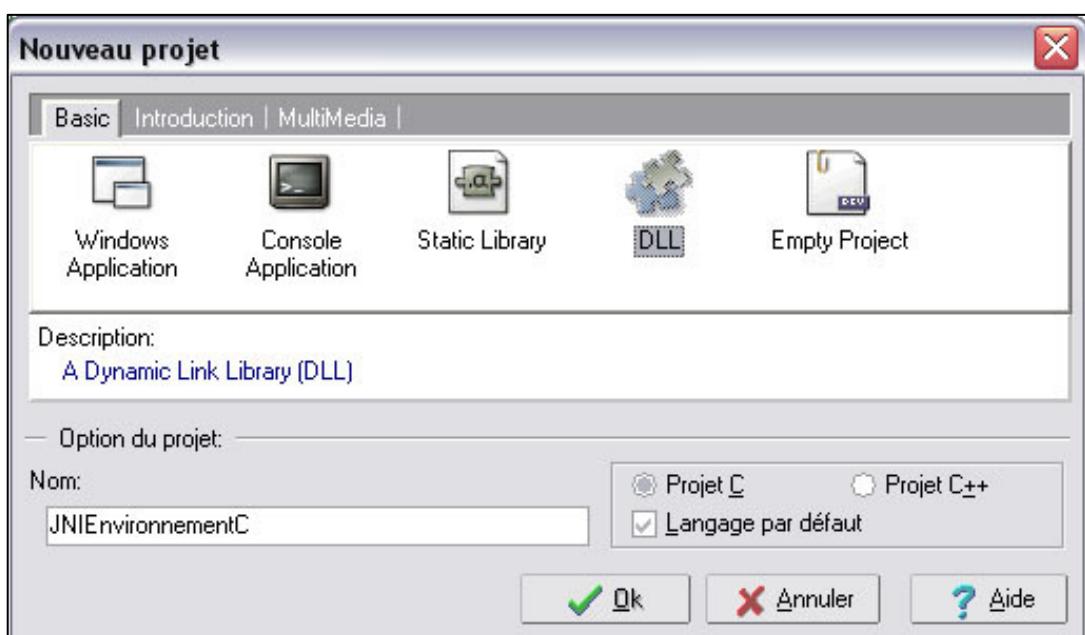
Nous savons à présent quel prototype donner à notre fonction. Par un moyen quelconque, créons donc un source C/C++ qui donnera naissance à une librairie partagée. Sous Windows, il s'agira donc d'une DLL.

ExploreEnvironnement.c

```
#include <stdlib.h>
#include <jni.h>
#include "jnienvironment_ExploreEnvironnement.h"

JNIEXPORT void JNICALL
Java_jnienvironment_ExploreEnvironnement_afficheVariablesEnv
(JNIEnv * env, jobject obj)
{
    printf("PATH = %s\n", getenv("PATH"));
    printf("PROMPT = %s\n", getenv("PROMPT"));
    return;
}
```

Nous construisons notre DLL avec un EDI quelconque, ici Dev-C++. Le classique projet Windows ressemblera donc à ceci :



```

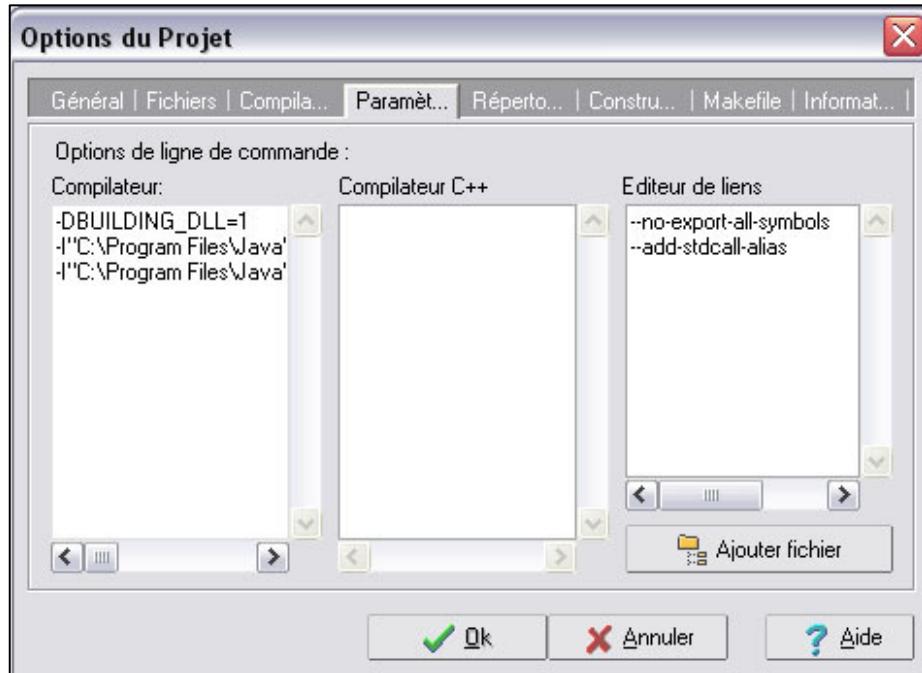
#include <stdlib.h>
#include <jni.h>
#include "jnienvironnement_ExploreEnvironnement.h"

JNIEEXPORT void JNICALL Java_jnienvironnement_ExploreEnvironnement_afficheVariablesEnv
    (JNIEnv * env, jobject obj)
{
    printf("PATH = %s\n", getenv("PATH"));
    printf("PROMPT = %s\n", getenv("PROMPT"));
    return;
}

```

Evidemment, il faut paramétriser le compilateur pour qu'il trouve le fichier jni.h (qui inclut lui-même le fichier jni_md.h) : on ajoutera donc pour le compilateur C :

-DBUILDING_DLL=1 -I"C:\Program Files\Java\jdk1.5.0_06\include" -I"C:\Program Files\Java\jdk1.5.0_06\include\win32"

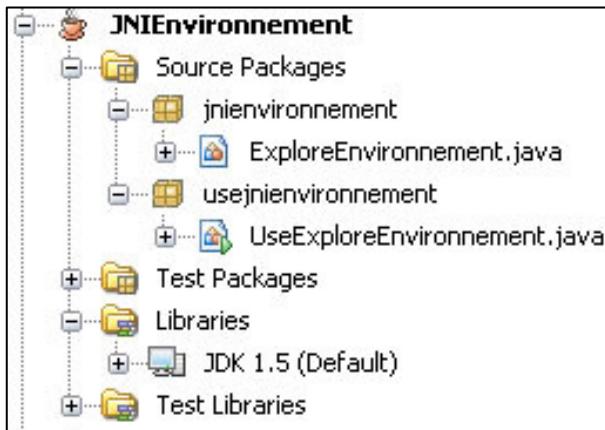


On obtient ainsi finalement après compilation et link un fichier **JNIEnvironnementC.dll**.
Sous UNIX-Sun Solaris, ce sera le triomphe de la ligne de commande :

```
%>cc -G -I/usr/local/java/include -I/usr/local/java/include/solaris ExploreEnvironnement.c
-o JNIEnvironnementC.so
```

6.5 L'exécution de l'application

Notre projet NetBeans a l'aspect suivant :



Cependant, il faut encore savoir que, sous Windows, la machine virtuelle de type Win32 utilise comme liste des chemins de recherche des librairies, outre le répertoire courant, les chemins indiqués dans la variable d'environnement PATH. Donc, sans précautions, un léger problème se révèle à l'exécution de notre application Java :

Echec au chargement de la librairie dynamiquejava.lang.UnsatisfiedLinkError: no **JNIEnvironnementC** in **java.library.path**
Java Result: 1

Donc, le plus simple pour éviter ceci est de placer la dll dans l'un de ces chemins, par exemple dans C:\Program Files\Java\jre1.5.0_06\bin. Un nouvel essai est cette fois concluant :

Système d'exploitation : Windows XP

PATH = C:\Program Files\Reflection; C:\WINDOWS\system32;C:\WINDOWS;
C:\WINDOWS\System32\Wbem;C:\Gemplus\GemXpresso.rad3\bin;C:\Gemplus\GEMXPR~
1.RAD\bin;C:\Program Files\MySQL\MySQL Server 5.0\bin;

PROMPT = (null)

Sous Unix, la démarche est similaire si l'on sait que la machine virtuelle utilise comme liste des chemins de recherche des librairies les chemins indiqués dans la variable d'environnement LD_LIBRARY_PATH.

Remarque

Les deux classes Java sont ici dans le même porjet NetBeans. Si ce n'est pas le cas, il faudra évidemment placer dans le projet de UseExploreEnvironnement une référence au jar contenant la classe ExploreEnvironnment.

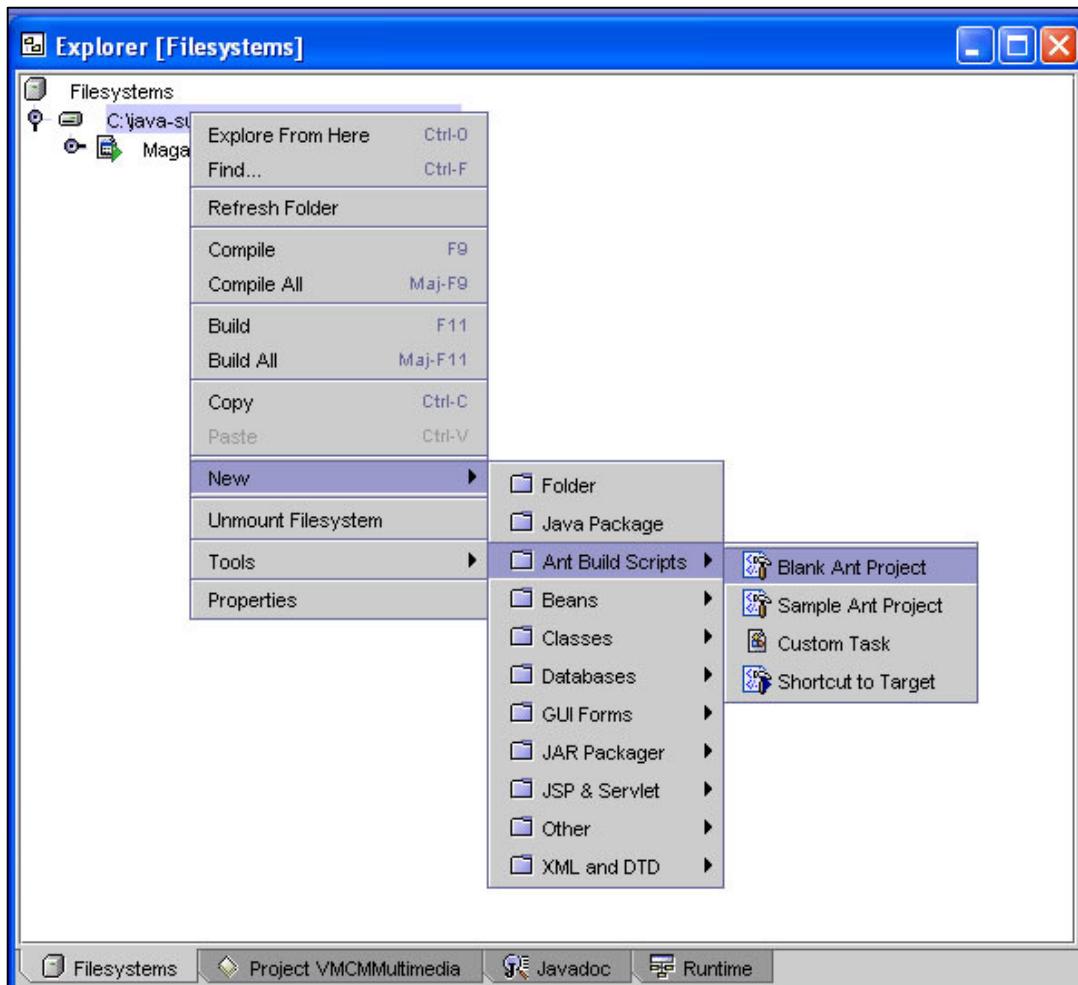
7. Les fichiers de commandes Ant

Tout qui a déjà programmé en C ou C++ (ou d'ailleurs en n'importe quoi d'autre) sous UNIX a sans doute utilisé l'utilitaire **make**, qui permet une compilation et une exécution "intelligente", c'est-à-dire tenant compte des dépendances des fichiers sources, objets et exécutables les uns envers les autres. Ce même utilisateur se souviendra que l'usage en est relativement délicat (du genre : un espace en trop ou à la place d'une tabulation et zut !) et, de toute manière, limité à l'environnement (en particulier, un shell) dans lequel on travaille.

Voilà évidemment qui n'arrange pas trop les développeurs Java qui ont constamment à l'esprit un souci de portabilité. C'est ci qu'intervient l'outil **ANT** (Another Neat Tool), produit par Apache Software Foundation dans le cadre du projet Jakarta⁶. ANT offre des possibilités de compilation similaires à celles d'un make, mais pour toute plate-forme parce qu'il se base sur XML⁷ et qu'il est écrit en Java : avec de tels outils, on fait ce que l'on veut ;-)



A titre d'exemple introductif, voyons comment réaliser un projet ANT depuis Netbeans. Démarrons à partir d'un fichier ANT quasi vide :



On obtient un fichier Ant/Xml de base (un "squelette"). Suivant les suggestions rencontrées ("changeme", "writeme"), nous pouvons écrire, pour obtenir la compilation des fichiers du

⁶ voir <http://ant.apache.org/>

⁷ une introduction à XML se trouve dans Java (III) - mais l'usage qui en est fait ici est rudimentaire.

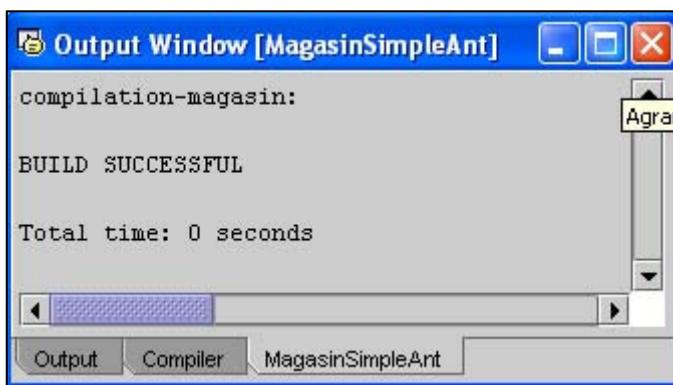
répertoire courant (ici, nous sommes le répertoires des exemples de containers développés au paragraphe 2) :

MagasinSimpleAnt.xml

```
<?xml version="1.0"?>

<project name="MagasinSimpleAnt" basedir=". " default="compilation-magasin">
    <target name="compilation-magasin">
        <javac deprecation="on" srcdir=". "/>
    </target>
</project>
```

Une simple exécution de fichier ANT donne :



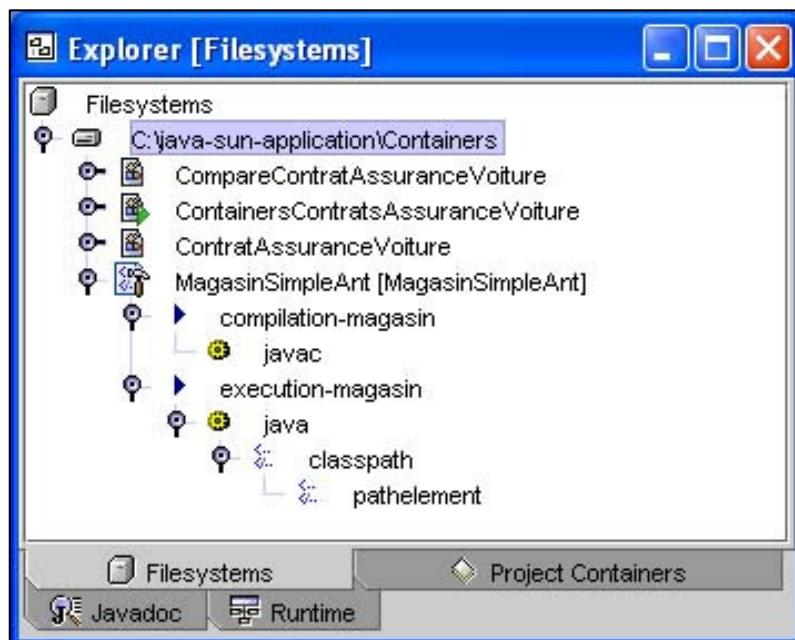
Pour ajouter l'exécution à la suite de la compilation, il suffit de créer un deuxième tag "target" :

MagasinSimpleAnt.xml (2)

```
<?xml version="1.0"?>

<project name="MagasinSimpleAnt" basedir=". " default="execution-magasin">
    <target name="compilation-magasin">
        <javac deprecation="on" srcdir=". "/>
    </target>
    <target name="execution-magasin" depends="compilation-magasin">
        <java classname="ContainersContratsAssuranceVoiture">
            <classpath>
                <pathelement location=". "/>
            </classpath>
        </java>
    </target>
</project>
```

Le fichier ANT montre clairement sa structure dans la fenêtre Filesystems :



Une exécution de ce fichier ANT conduit à l'exécution de l'ensemble de l'application. Dans les anciennes versions de l'EDI (comme Java Forte), il faut cependant positionner à true l'attribut fork de la balise <java> dans sa liste de propriétés :

Property	Value
args	
classname	Magasin
classpath	
classpathref	
dir	
failonerror	False
fork	True
ID	True
jar	False
jvm	
jvmargs	
jvmversion	
maxmemory	
output	

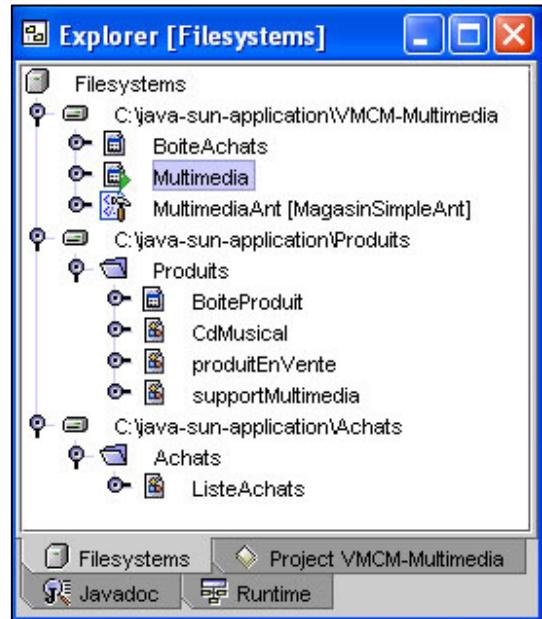
ainsi, l'exécution demandée est réalisée sur une nouvelle machine virtuelle, la JVM initiale restant celle sur laquelle le fichier ant est exécuté. Le fichier xml voit ainsi sa balise <java> complétée selon :

```
<java classname=" ContainersContratsAssuranceVoiture " fork="true">
```

Enfin, on aura encore remarqué que, si on utilise des packages, il suffit d'utiliser

- ◆ pour javac, l'attribut **classpath**;
- ◆ pour java, l'attribut **classpath** analogue ou le tag composite **<classpath>** qui comporte un ou plusieurs tags **<pathelement>**.

Ainsi, dans le cadre d'une application VMCM-Multimedia, si on utilise des packages Produits et Achats dont les répertoires ont été montés :

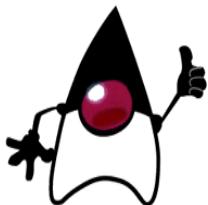


le fichier ant correspondant sera :

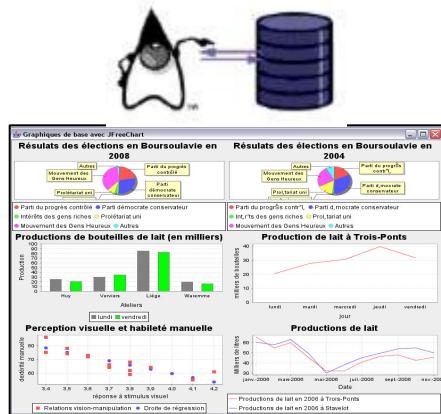
MultimediaAnt.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project basedir=". " default="execution-multimedia" name="MagasinSimpleAnt">
    <target name="compilation-multimedia">
        <javac deprecation="on" srcdir=". " classpath="c:\java-sun-
application\Produits;c:\java-sun-application\Achats" />
    </target>
    <target depends="compilation-multimedia" name="execution-multimedia">
        <java classname="Multimedia" fork="true">
            <classpath>
                <pathelement location=". "/>
                <pathelement location="c:\java-sun-application\Produits"/>
                <pathelement location="c:\java-sun-application\Achats"/>
            </classpath>
        </java>
    </target>
</project>
```

Utilitaires particuliers : ceux qui tournent autour des statistiques. Ben oui, sans doute est-il temps de nous tourner vers les chiffres ☺ ...



XV. La librairie JFreeChart



Je m'aime trop moi-même pour pouvoir haïr qui que ce soit.

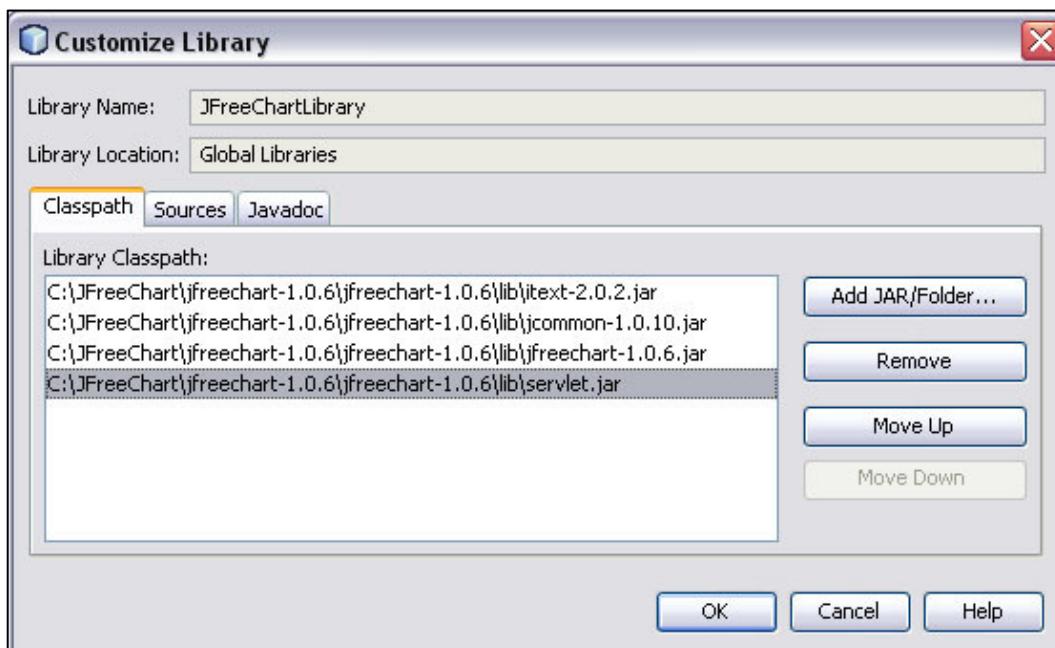
(J.-J. Rousseau, Les rêveries du promeneur solitaire)

[Un grand merci à J-M Wagner qui a bien débroussaillé le terrain à l'occasion d'une micro-formation de la Java Team ☺ !]

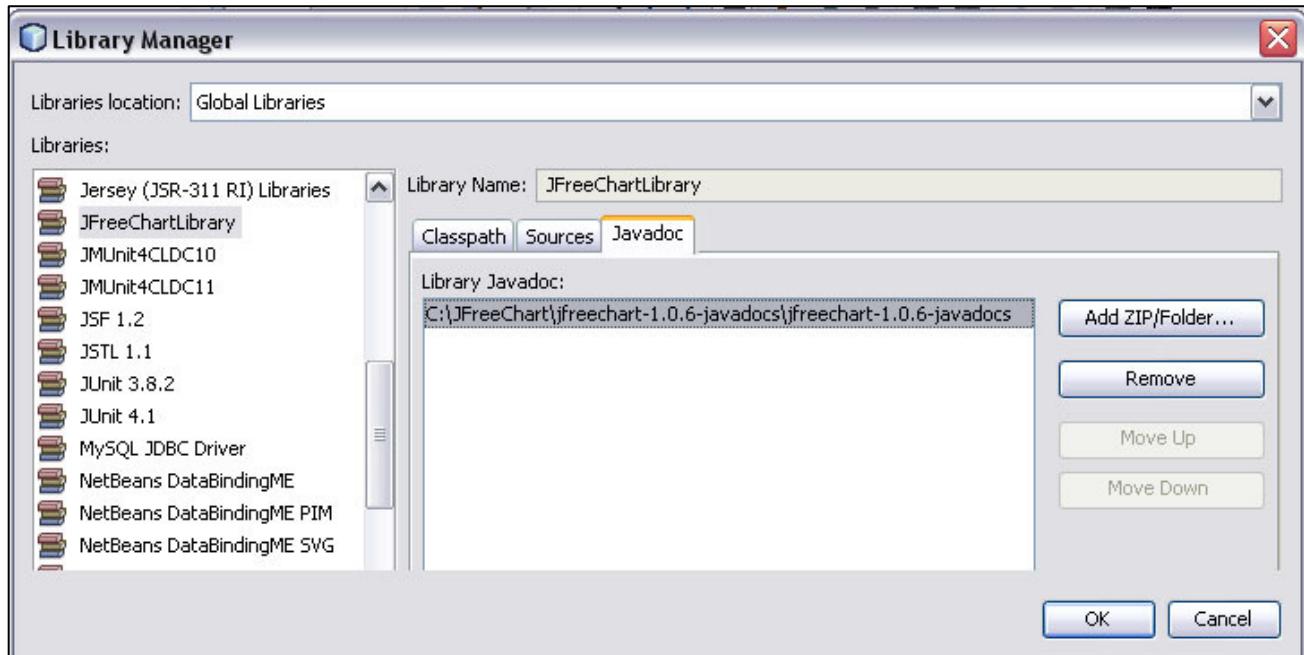
1. Une bibliothèque pour graphiques statistiques

JFreeChart est une librairie de classes Java offrant un large éventail de fonctionnalités de graphiques statistiques (dans un sens très large) utilisables au sein d'une application, d'une applet ou d'une servlet. Développée en 2000, elle est distribuée gratuitement (par sourceforge.net), avec son code source, sous licence GNU-LGPL, c'est-à-dire qu'elle peut être utilisée dans une application quelconque, libre ou propriétaire. On peut trouver les informations sur le software à <http://www.jfree.org/jfreechart/>, où l'on peut également le télécharger sous forme d'un fichier zip (jfreechart-1.0.6.zip) ainsi que la documentation correspondante.

On s'en doute, un certain nombre de jars matérialisent la librairie. Plutôt que de les incorporer un par un dans NetBeans 6.*, nous créons une librairie JFreeChartLibrary qui contient les jars utiles pour un usage courant. Classiquement, un clic droit sur le nœud Libraries du projet, "Add library ..." puis "Create ..." puis "Create new library" conduit à :



Un dernier "Add library" permet d'effectuer l'ajout de cette librairie au projet. On peut vérifier l'existence de notre nouvelle librairie par Tools→Libraries dans le menu principal et aussi en profiter pour y adjoindre les javadocs correspondants :



Nous voilà donc fin prêts à travailler ...

2. La représentation des données

L'idée de base est que les données destinées à produire un graphique statistique sont placées dans un container (un "**dataset**") implémentant l'interface **Dataset** : celui-ci se borne à assurer qu'un mécanisme d'événement (au sens des Java Beans) existe pour notifier tout modification de données au sein du dataset. On trouve donc des méthodes comme :

```
void addChangeListener (DatasetChangeListener listener)
```

où, dans la logique habituelle des Java Beans, le listener est un objet implémentant un interface **DatasetChangeListener** dont la seule méthode est :

```
void datasetChanged (DatasetChangeEvent event)
```

tandis que l'objet événement est capable de fournir le dataset concerné grâce à la méthode :

```
public Dataset getDataset()
```

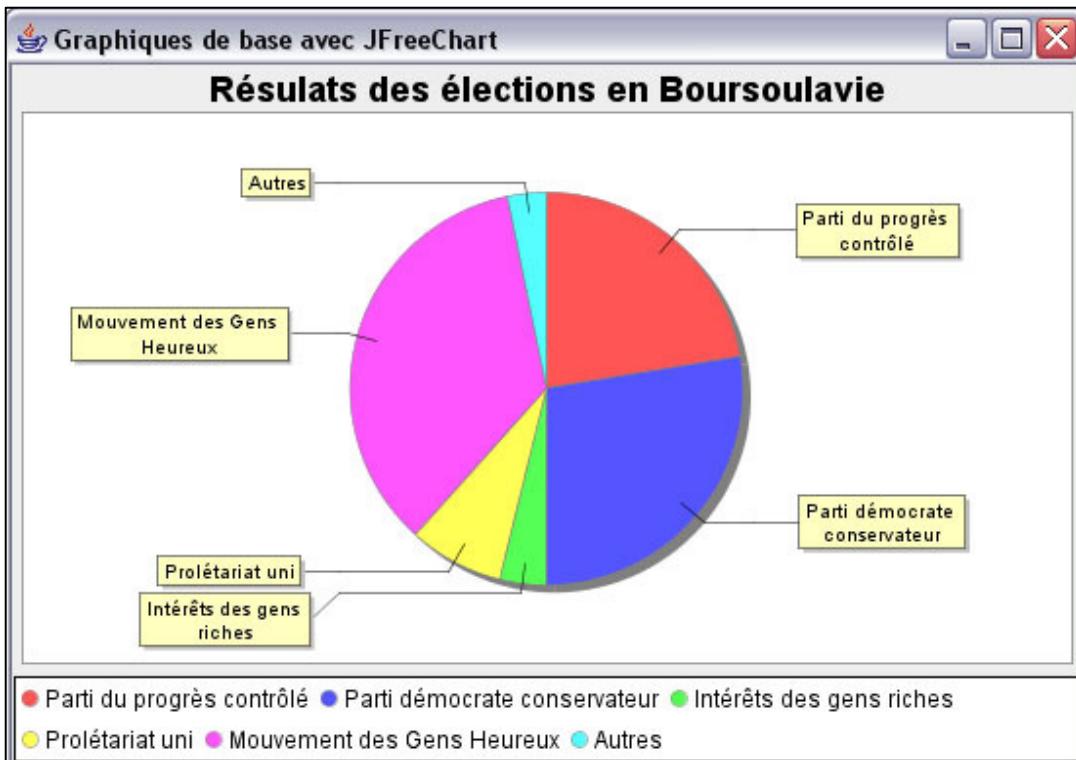
La manière de définir les données contenues dans le dataset dépend bien sûr du type de graphique statistique auquel il est destiné à servir : sectoriel, histogramme, nuage de points, etc. La classe abstraite **AbstractDataset**, qui implémente l'**interface**, ne pipe donc mot de méthodes d'insertion ou de retrait, mais focalise toujours sur la logique événementielle, avec des méthodes **protected** comme :

```
protected void fireDatasetChanged()
protected void notifyListeners(DatasetChangeEvent event)
```

Ne se croirait-on pas revenus dans Java I (à la fin ...) ;-)

3. L'exemple classique : le diagramme sectoriel

Afin d'illustrer la suite de notre propos, nous allons réaliser un diagramme sectoriel du type suivant :



3.1 Un Dataset particulier

Ce type de graphique est encore familièrement appelé "diagramme en camembert" ou "tarte" – *pie* en anglais. A priori, une manière de fournir les données consisterait à fournir les libellés des données (ici, le nom des partis politiques) et le pourcentage associé : c'est le rôle de la classe **DefaultPieDataset** qui, outre le fait qu'elle implémente la classe abstraite **AbstractDataset**, implémente aussi un interface **PieDataset** qui décrit les méthodes d'ajout de données dans un dataset pour "camembert" :

```
public void setValue (Comparable key, double value)
public void insertValue (int position, Comparable key, double value)
```

Comme on l'aura compris, ce genre de dataset est un container associatif de type hast-table, puisque la notion de clé est explicitement visible. Dans le cas de la deuxième méthode, le fait d'utiliser une clé existante a pour effet de réaliser une mise à jour de la valeur associée avant déplacement à la position souhaitée (comptée à partir de 0). Dans les deux cas, un événement **DatasetChangeEvent** sera envoyé aux listeners intéressés.

Bien sûr, la méthode

```
public java.lang.Number getValue (java.lang.Comparable key)
```

est aussi prévue. Dans le cas de notre exemple, les choses sont donc aussi simples que ceci :

```
DefaultPieDataSet ds = new DefaultPieDataSet(); // contient les data  
ds.setValue("Parti du progrès contrôlé", 22.36);  
ds.setValue("Parti démocrate conservateur", 27.69);  
ds.setValue("Intérêts des gens riches", 3.78);  
ds.setValue("Prolétariat uni", 7.85);  
ds.setValue("Mouvement des Gens Heureux", 35.12);  
ds.setValue("Autres", 3.2);
```

Donc, "Prolétariat uni" (par exemple) est considéré comme une clé qui permet d'accéder à l'information "7.85". Les données étant définies, il nous faut un objet réalisant le dessin souhaité ...

3.2 Le JFreeChart correspondant

La classe **JFreeChart** représente en fait un graphique statistique au sens général (histogramme, linéaire, sectoriel, nuage de points, ...). Le graphique réalisé sera dessiné en utilisant l'API Java2D. Un tel objet réalise ce travail en utilisant trois variables membres :

- ◆ une instance de **Dataset**, qui contient évidemment les données;
- ◆ une instance d'une classe dérivée de la classe abstraite **Plot**, qui gère notamment la représentation des données et leur référentiel (comme des axes);
- ◆ une instance dérivée de **Title**, qui comporte en particulier la légende du graphique).

Il existe bien sûr des constructeurs prévisibles comme

```
public JFreeChart (Plot plot)  
public JFreeChart (java.lang.String title, Plot plot)
```

- mais nous passerons plutôt par une factory ...

Signalons encore que la classe implémente les listeners **TitleChangeListener** et **PlotChangeListener** et qu'elle peut utiliser les services d'un **ChartChangeListener** :

```
public void addChangeListener (ChartChangeListener listener)  
public void fireChartChanged ()
```

à la sémantique assez évidente.

3.3 La classe factory

La classe **ChartFactory** regroupe en fait une série de méthodes permettant de créer un JFreeChart de type "camembert", "histogramme", "nuage de points", etc avec une série d'attributs par défaut (mais reconfigurables) : le rôle de ces "factories" n'est donc pas tant une question de portabilité que de facilité ... Citons parmi elles celle qui nous intéressera pour notre exemple :

```
public static JFreeChart createPieChart (java.lang.String title, PieDataset dataset,  
boolean legend, boolean tooltips, boolean urls)
```

les trois derniers paramètres indiquant si l'on souhaite une légende, des "tooltips" et la possibilité d'hyperliens (mais il faudrait alors les définir). Donc, pour notre exemple :

```
JFreeChart jf = ChartFactory.createPieChart("Résultats des élections en Boursoulavie", ds,  
true, true, false);
```

3.4 Les Plots

En fait, un objet JFreeChart possède une variable membre qui est une implémentation de l'interface **Plot** : son rôle est de réaliser les dessins relativement aux axes et aux données. Le JFreeChart utilisé ci-dessus, parce qu'il a été créé avec la factory `createPieChart()`, s'est vu muni automatiquement d'un objet instance de la classe dérivée **PiePlot**. On peut bien entendu définir un objet Plot explicitement pour ensuite l'affecter au service de notre JFreeChart ou encore configurer le Plot existant, par exemple ainsi :

```
PiePlot pp = (PiePlot)jfc.getPlot();  
pp.setSectionPaint("Prolétariat uni", new Color(255,0,0)); // logique, non ?
```

En fait, l'objet Plot contient un **Renderer** qui permet de configurer les séries de couleurs, etc ...

Comme d'habitude, un objet Plot peut se faire escorter de listeners :

```
public void addChangeListener(PlotChangeListener listener)  
public void notifyListeners(PlotChangeEvent event)
```

et déclare des méthodes prévisibles :

```
public void setOutlineVisible(boolean visible)  
public void setBackgroundImage(java.awt.Image image)  
...
```

La classe dérivée PiePlot possède ses méthodes propres comme :

```
public void setSectionPaint(Comparable key, Paint paint)  
public void setStartAngle(double angle)
```

Pour cette dernière méthode, tout est horlogique par défaut : le point de départ est à 90° et dans le sens négatif d'un point de vue trigonométrique.

3.5 Le composant container visuel

L'objet JFreeChart ainsi construit peut maintenant être placé dans un composant graphique permettant son affichage, donc un objet de type `JFrame` ou `JPanel`. La librairie JFreeChart fournit des classes dérivées des classes de Swing qui réalisent le travail d'intégration du graphique : `ChartFrame` et `ChartPanel`. Les constructeurs attendus sont bien disponibles :

```
public ChartFrame(java.lang.String title, JFreeChart chart)  
public ChartFrame(java.lang.String title, JFreeChart chart, boolean scrollPane)
```

et l'on peut donc imaginer dans l'application support :

```
ChartFrame fr = new ChartFrame("Résultats", jfc);
```

```
fr.pack();
fr.setVisible(true)
```

Cependant, le plus souvent, ce sera un panel qui sera l'élément de visualisation :

```
public ChartPanel (JFreeChart chart)
```

en remarquant, en passant, des méthodes alléchantes comme

```
public void doSaveAs () throws java.io.IOException
public void createChartPrintJob ()
```

Mais bref :

```
ChartPanel cp = new ChartPanel(jfc);
setContentPane(cp);
```

3.6 L'application complète

En résumé :

FenAppChart.java

```
/*
 * FenAppChart.java
 */

package jfreechartconcepts;

import java.awt.Panel;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import org.jfree.chart.ChartFactory;
import org.jfree.chart.ChartPanel;
import org.jfree.chart.JFreeChart;
import org.jfree.data.general.DefaultPieDataset;

/**
 * @author Vilvens
 */

public class FenAppChart extends javax.swing.JFrame
{
    public FenAppChart()
    {
        initComponents();
        showPieChart();
    }

    private void initComponents() { ... }
```

```

private void showPieChart()
{
    // 1. Définir un dataset qui contient les data
    DefaultPieDataset ds = new DefaultPieDataset();
    ds.setValue("Parti du progrès contrôlé", 22.36);
    ds.setValue("Parti démocrate conservateur", 27.69);
    ds.setValue("Intérêts des gens riches", 3.78);
    ds.setValue("Prolétariat uni", 7.85);
    ds.setValue("Mouvement des Gens Heureux", 35.12);
    ds.setValue("Autres", 3.2);

    // 2. Se fournir un JFreeChart
    JFreeChart jfc = ChartFactory.createPieChart (
        "Résultats des élections en Boursoulavie", ds, true, true, true);

    // 3. Fabriquer le Panel
    ChartPanel cp = new ChartPanel(jfc);
    setContentPane(cp);
}

public static void main(String args[]) {
    java.awt.EventQueue.invokeLater(new Runnable() {
        public void run() {
            new FenAppChart().setVisible(true);
        }
    });
}
}

```

Le résultat est bien celui annoncé en début de paragraphe ☺

3.7 La cascade des événements

Il importe de bien remarquer le fonctionnement interne des éléments de la librairie, peu manifeste ici parce que le résultat est affiché en une seule fois. En fait, à chaque ajout de donnée au Dataset, une succession d'événements "à la Java Beans" sont engendrés :

- ➔ **DatasetChangeEvent** : reçu par l'objet Plot qui est DatasetChangeListener; il émet ...
- ➔ **PlotChangeEvent** : reçu par l'objet Chart qui est PlotChangeListener; il émet ...
- ➔ **ChartChangeEvent** : reçu par l'objet ChartPanel qui est ChartChangeListener; il redessine l'entièreté du dessin (ce qui n'est pas très performant dans le cas d'animations, mais bon).

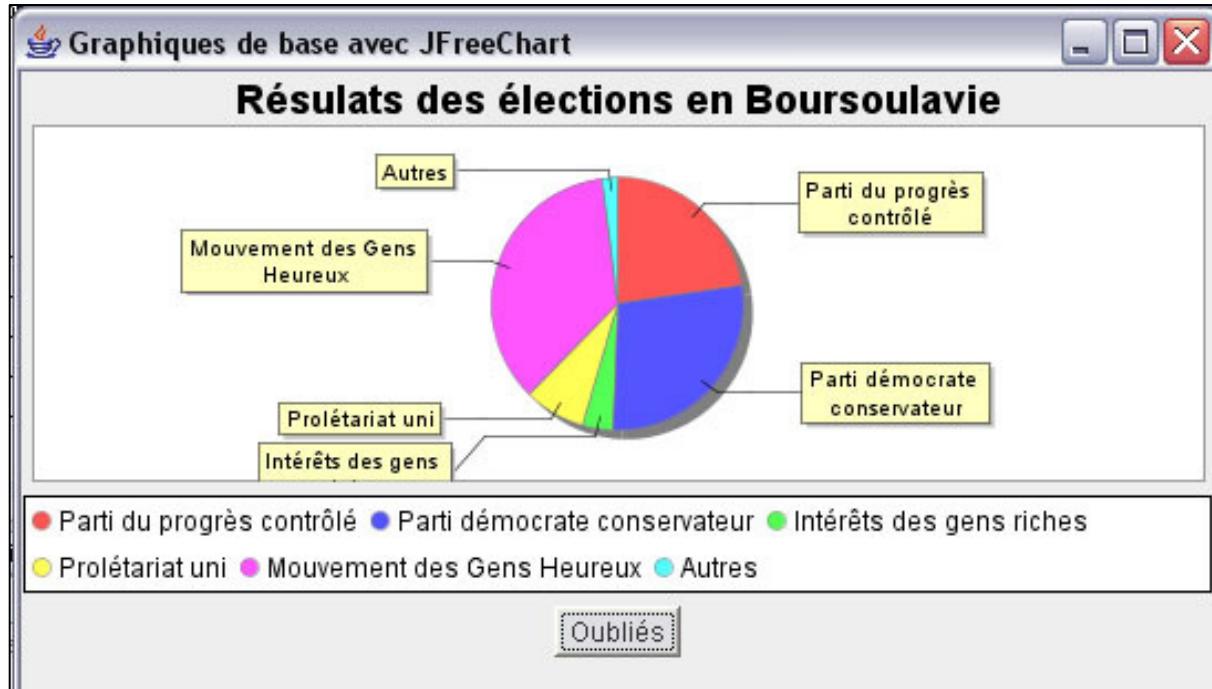
Donc, si on ajoute à la fenêtre de l'application un bouton "Oubliés" et que l'événement Action de ce bouton a pour effet d'ajouter une donnée au Dataset :

```

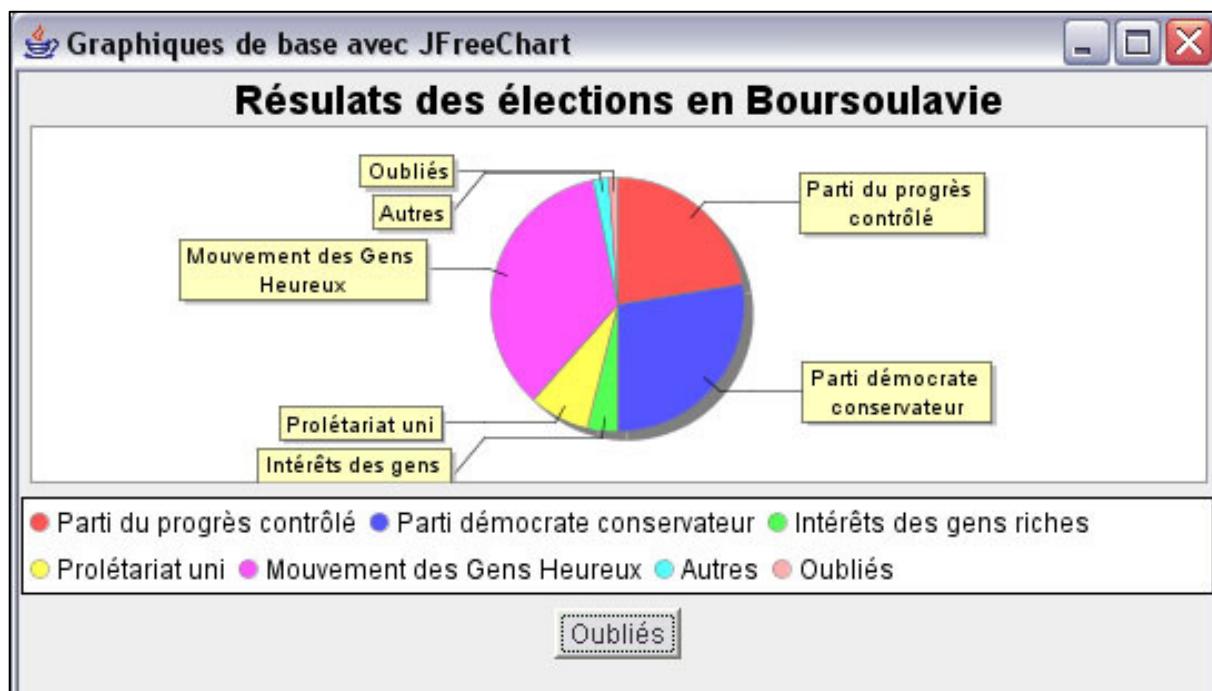
public void actionPerformed(ActionEvent e)
{
    ds.setValue("Oubliés", 1.2); // ds est devenu une variable membre
}

```

l'appui sur le bouton aura pour effet de mettre le graphique à jour. Ainsi, si d'abord :

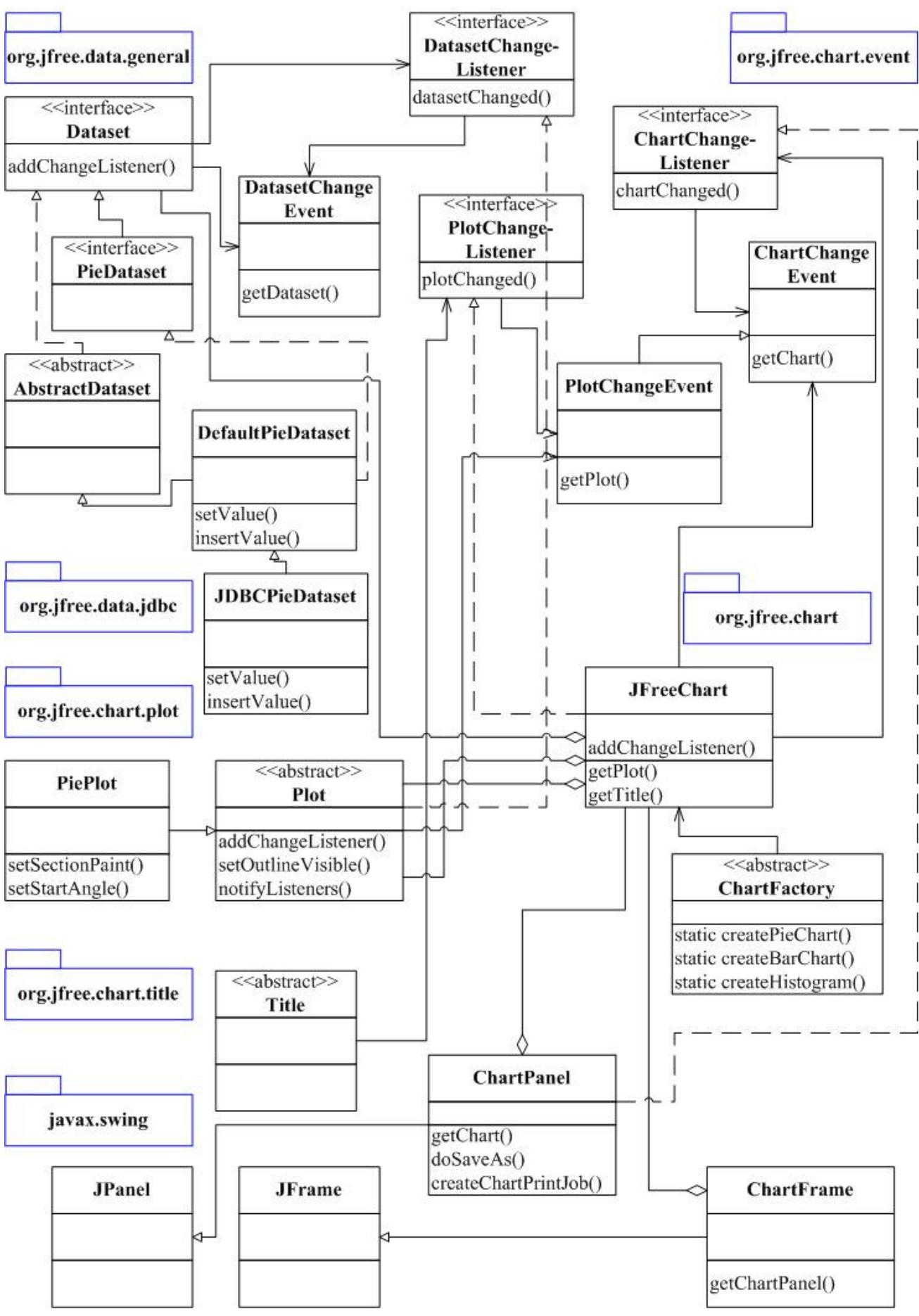


on aura, après appui :



3.8 Le diagramme de classes UML

Sans doute n'est-il pas superflu de visualiser les relations entre les différents intervenants :



4. L'utilisation d'une base de données

4.1 Une classe facilitatrice

Un lecteur attentif aura remarqué la présence d'une classe non encore évoquée : **JDBCPieDataset**, dérivée de DefaultPieDataset. Comme on l'aura compris, il est possible de créer un graphique statistique à partir d'une base de données et la librairie fournit la classe qui facilite ce travail (nous pourrions nous en passer et accéder aux données par nos propres moyens, mais quel intérêt ?), en l'occurrence **JDBCPieDataset** qui matérialise un dataset à connecter sur une base de données.

Mais campons tout d'abord le décor ...

4.2 Les données dans une base de données

Avec derrière la tête l'idée de comparer les élections de 2008 à celles de 2004, exécutons les instructions SQL suivantes avec MySQL :

```
mysql> create database Elections;
mysql> use Elections;
mysql> grant all privileges on Elections to 'CFBUNAT'@'localhost' identified by
'ViveLeParti' with grant option;
mysql> create table Boursoulavie2004 (parti VARCHAR(30), pourcents FLOAT);
mysql> grant all privileges on Boursoulavie2004 to 'CFBUNAT'@'localhost' identified by
'ViveLeParti' with grant option;
mysql> insert into Boursoulavie2004 values ('Parti du progrès contrôlé', 18.25);
mysql> insert into Boursoulavie2004 values ('Parti démocrate conservateur', 32.09);
mysql> insert into Boursoulavie2004 values ('Intérêts des gens riches', 1.52);
mysql> insert into Boursoulavie2004 values ('Prolétariat uni', 15.77);
mysql> insert into Boursoulavie2004 values ('Mouvement des Gens Heureux', 24.43);
mysql> insert into Boursoulavie2004 values ('Autres', 7.94);

mysql> select * from Boursoulavie2004;
+-----+-----+
| parti | pourcents |
+-----+-----+
| Parti du progrès contrôlé | 18.25 |
| Parti démocrate conservateur | 32.09 |
| Intérêts des gens riches | 1.52 |
| Prolétariat uni | 15.77 |
| Mouvement des Gens Heureux | 24.43 |
| Autres | 7.94 |
+-----+-----+
6 rows in set (0.02 sec)
```

L'utilisateur créé peut effectivement agir :

```
C:\>mysql -u CFBUNAT -pViveLeParti
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 9
```

Server version: 5.0.41-community-nt MySQL Community Edition (GPL)

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

```
mysql> use Elections;
Database changed
mysql> select * from Boursoulavie2004;
+-----+-----+
| parti | pourcents |
+-----+-----+
| Parti du progrès contrôlé | 18.25 |
| Parti démocrate conservateur | 32.09 |
| Intérêts des gens riches | 1.52 |
| Prolétariat uni | 15.77 |
| Mouvement des Gens Heureux | 24.43 |
| Autres | 7.94 |
+-----+-----+
6 rows in set (0.03 sec)

mysql>
```

4.3 Le diagramme sectoriel à partir d'une base de données

L'utilisation de la classe **JDBC PieDataset** est très simple :

1) on crée une connexion (classe Connection de java.sql) vers la base de donnée souhaitée – le plus souvent, un Java Bean existe déjà pour réaliser ce travail;

2) on instancie un JDBC PieDataset sur cette connexion en utilisant le constructeur :

```
public JDBC PieDataset (Connection con)
```

Le Dataset ainsi créé est vide (car il se pourrait que l'on puisse le remplir avec différentes données, à choisir dynamiquement).

3) on remplit [*populate*] le Dataset au moyen de sa méthode :

```
public void executeQuery (String query) throws java.sql.SQLException
```

En adaptant notre programme précédent, cela donne donc :

FenAppChart.java (2)

```
/*
 * FenAppChart.java
 */
package jfreechartconcepts;

import com.mysql.jdbc.Connection;
import java.awt.GridLayout;
import java.sql.DriverManager;
import java.sql.SQLException;
```

```
import org.jfree.chart.ChartFactory;
import org.jfree.chart.ChartPanel;
import org.jfree.chart.JFreeChart;
import org.jfree.data.general.DefaultPieDataset;
import org.jfree.data.jdbc.JDBCPieDataset;

/**
 * @author Vilvens
 */

public class FenAppChart extends javax.swing.JFrame
{
    public FenAppChart()
    {
        initComponents();
        showPieChart();
        showPieChartJdbc();
    }

    private void initComponents() { ... }

    public static void main(String args[])
    {
        java.awt.EventQueue.invokeLater(new Runnable()
        {
            public void run()
            {
                new FenAppChart().setVisible(true);
            }
        });
    }

    private void showPieChart()
    {
        // 1. Définir un dataset qui contient les data
        //DefaultPieDataset
        DefaultPieDataset ds = new DefaultPieDataset();
        ds.setValue("Parti du progrès contrôlé", 22.36);
        ...

        // 2. Se fournir un JFreeChart
        JFreeChart jfc = ChartFactory.createPieChart(
            "Résultats des élections en Boursoulavie en 2008", ds, true, true, true);

        // 3. Fabriquer le Panel
        ChartPanel cp = new ChartPanel(jfc);

        this.getContentPane().setLayout(new GridLayout(2,1));
        this.getContentPane().add(cp);
    }
}
```

```

private void showPieChartJdbc()
{
    JDBCPIEDataset jds = null;

    String url = "jdbc:mysql://localhost:3306/Elections";
    try
    {
        Class.forName("com.mysql.jdbc.Driver");
    }
    catch (ClassNotFoundException e)
    {
        System.err.println(e.getMessage());
    }
    System.out.println("Driver MySQL (com) chargé");

    Connection con = null;
    try
    {
        con = (Connection) DriverManager.getConnection(
            url, "CFBUNAT", "ViveLeParti");
        System.out.println("Connexion à la BDD Elections réalisée");
        jds = new JDBCPIEDataset(con);
        String req = "SELECT * FROM Boursoulavie2004;";
        jds.executeQuery(req);
        System.out.println("Dataset chargé");
        con.close();
    }
    catch (SQLException e)
    {
        System.err.println(e.getMessage());
    }
    catch (Exception e)
    {
        System.err.println(e.getMessage());
    }

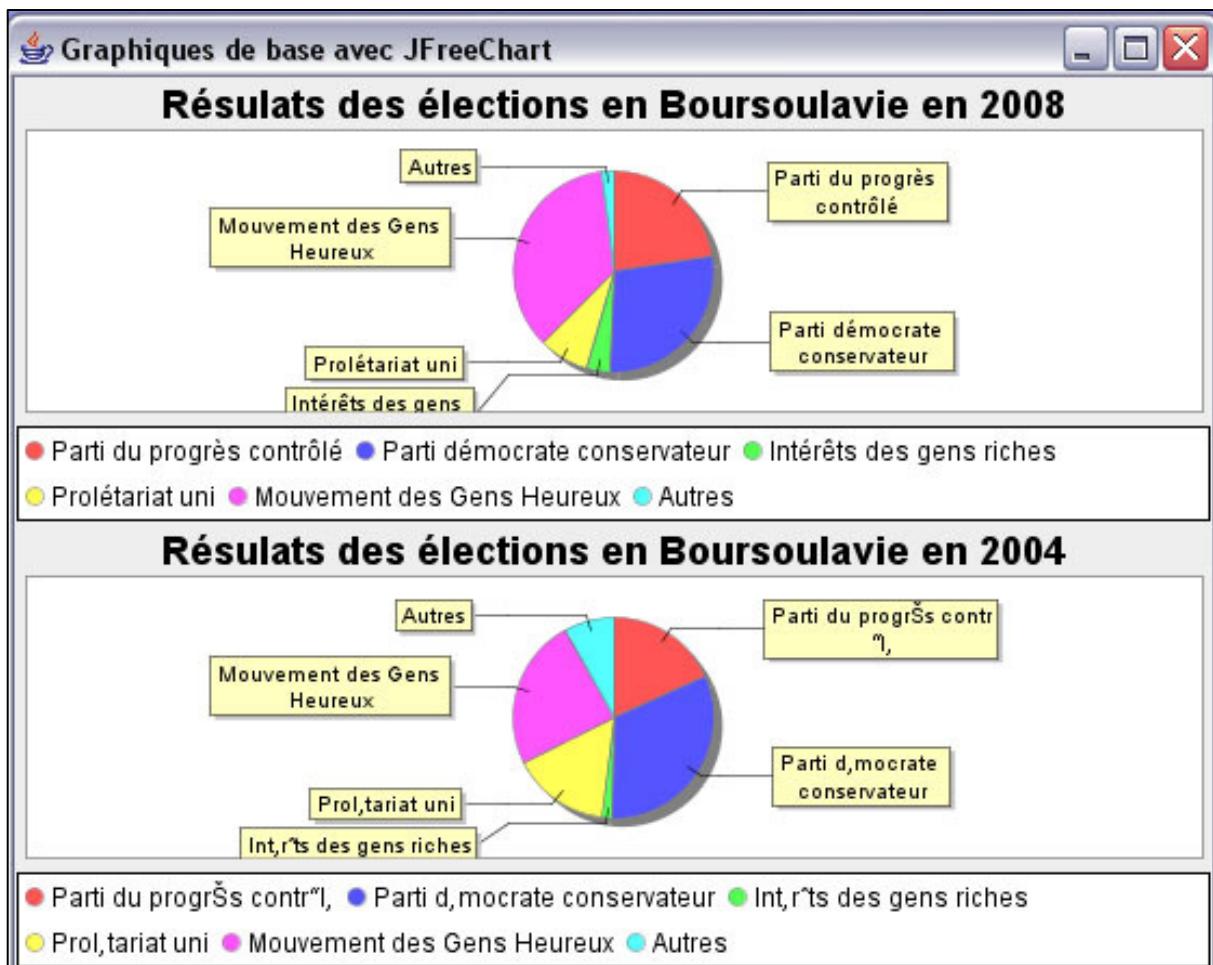
    JFreeChart jfcbd = ChartFactory.createPieChart(
        "Résultats des élections en Boursoulavie en 2004",
        jds,
        true, // générer des légendes ?
        true, // générer des tooltips ?
        true); // générer des URLs?

    // 3. Fabriquer le Panel
    ChartPanel cpbd = new ChartPanel(jfcbd);

    this.getContentPane().add(cpb);
}
}

```

avec pour résultat :



Tiens, tiens ... Un petit problème de charset ;-);-)?

On peut encore savoir qu'il existe des versions polymorphes du constructeur :

```
public JDBCPieDataset (String url, String driverName, String user, String password)
    throws java.sql.SQLException, java.lang.ClassNotFoundException
public JDBCPieDataset (Connection con, String query)
    throws java.sql.SQLException
```

ainsi que de la méthode d'exécution de la requête :

```
public void executeQuery (Connection con, String query)
    throws java.sql.SQLException
```

Evidemment, la première forme du constructeur dissimule l'objet Connection et, dans ce cas, il faut utiliser la méthode

```
public void close()
```

qui permet de refermer celle-ci.

5. D'autres graphiques classiques

5.1 Les histogrammes comparés

Après le diagramme sectoriel, les histogrammes sont les graphiques les plus populaires, en même temps que les graphiques d'évolution ("lignes brisées"). On s'en doute, la classe **ChartFactory** est capable de les générer avec par exemple :

```
public static JFreeChart createHistogram(String title,  
                                         String xAxisLabel,  
                                         String yAxisLabel,  
                                         IntervalXYDataset dataset,  
                                         PlotOrientation orientation,  
                                         boolean legend,  
                                         boolean tooltips,  
                                         boolean urls)
```

Mais le plus souvent, il s'agit de comparer les histogrammes (les "bar charts") et nous utiliserons d'emblée :

```
public static JFreeChart createBarChart(String title,  
                                         String categoryAxisLabel,  
                                         String valueAxisLabel,  
                                         CategoryDataset dataset,  
                                         PlotOrientation orientation,  
                                         boolean legend,  
                                         boolean tooltips,  
                                         boolean urls)
```

Le dataset à utiliser reste ici un **DefaultCategoryDataset**, dont la filiation est calquée sur celle des **DefaultPieDataset** : elle implémente la classe abstraite **AbstractDataset** qui elle-même implémente l'interface **Dataset**. Il est caractérisé par le fait que les valeurs qu'il contient (qui fixeront les valeurs en Y) dépendent de *deux paramètres* (on peut encore parler de *clés*) : celui qui détermine les groupes de rectangles de l'histogramme comparé et celui qui est à porter en X. Ainsi, par exemple, si nous nous intéressons aux productions de bouteilles de lait par jour dans un groupe de laiteries (disons que divers ateliers répartis géographiquement font partie de la même entreprise), ce type de graphique est idéal puisque les deux clés sont le jour et l'atelier. Un tel Dataset est instancié vide, puis se construit au moyen de la méthode :

```
public void addValue(double value, Comparable rowKey, Comparable columnKey)
```

Les deux derniers paramètres correspondent bien sûr aux deux paramètres évoqués plus haut : on peut considérer que l'on peut représenter les données dans une matrice, ce qui explique les noms données à ces paramètres dans la documentation ... Ce ne sont pas forcément des chaînes de caractères : on leur demande juste d'être **Comparable** (ce qui n'exige que la méthode **compareTo()**).

Sans finasser, supposons que nous créons le Dataset suivant :

```
String jour1 = "lundi", jour2 = "vendredi";  
String atelier1 = "Huy", atelier2 = "Verviers", atelier3 = "Liège", atelier4 = "Waremme";
```

```
DefaultCategoryDataset ds = new DefaultCategoryDataset();

ds.addValue(25.3, jour1, atelier1);
ds.addValue(20.7, jour2, atelier1);

ds.addValue(30.1, jour1, atelier2);
ds.addValue(34.2, jour2, atelier2);

ds.addValue(85.3, jour1, atelier3);
ds.addValue(82.1, jour2, atelier3);

ds.addValue(19.6, jour1, atelier4);
ds.addValue(15.9, jour2, atelier4);
```

Comme pour un PieChart, il nous suffit à présent de créer un BarChart par :

```
JFreeChart jfc = ChartFactory.createBarChart(
    "Productions de bouteilles de lait (en milliers)",
    "Ateliers",
    "Production",
    ds,
    PlotOrientation.VERTICAL,
    true, true, false
);
```

On remarquera, outre l'apparition de titres pour les axes, l'utilisation de la constante indiquant l'orientation du graphique, membre statique de la classe **PlotOrientation** qui se limite à cela :

```
public static final PlotOrientation HORIZONTAL
public static final PlotOrientation VERTICAL
```

L'application minimale suivante :

FenAppChart.java (3)

```
/*
 * FenAppChart.java
 */

package jfreechartconcepts;

import com.mysql.jdbc.Connection;
import java.awt.GridLayout;
import java.sql.DriverManager;
import java.sql.SQLException;
import org.jfree.chart.ChartFactory;
import org.jfree.chart.ChartPanel;
import org.jfree.chart.JFreeChart;
import org.jfree.chart.plot.PlotOrientation;
import org.jfree.data.category.DefaultCategoryDataset;
```

```

/**
 * @author Vilvens
 */

public class FenAppChart extends javax.swing.JFrame
{
    public FenAppChart()
    {
        initComponents();
        getContentPane().setLayout(new GridLayout(1,1));
        showHistogram();
    }

    private void initComponents() { ... }

    private void showHistogram()
    {
        String[] jours = {"lundi", "vendredi"};
        String[] ateliers = {"Huy", "Verviers", "Liège", "Waremme"};

        double[][] valProduction = { {25.3, 20.7}, {30.1, 34.2}, {85.3, 82.1}, {19.6, 15.9} };

        DefaultCategoryDataset ds = new DefaultCategoryDataset();

        for (int i=0; i<ateliers.length; i++)
            for (int j=0; j< jours.length; j++)
                ds.addValue(valProduction[i][j], jours[j], ateliers[i]);

        JFreeChart jfc = ChartFactory.createBarChart(
            "Productions de bouteilles de lait (en milliers)",
            "Ateliers",
            "Production",
            ds,
            PlotOrientation.VERTICAL,
            true, true, false
        );

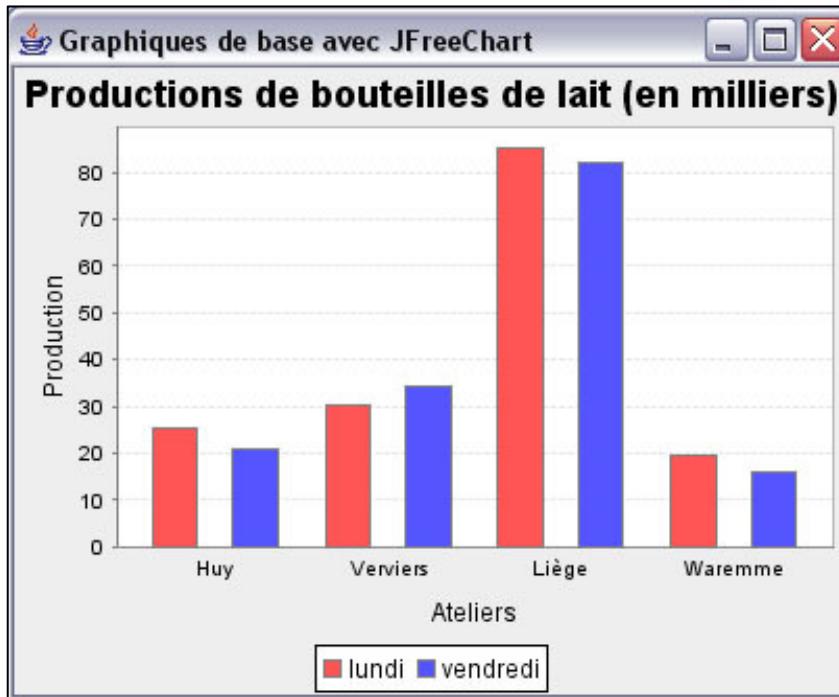
        ChartPanel cp = new ChartPanel(jfc);

        this.getContentPane().add(cp);
    }

    public static void main(String args[])
    {
        java.awt.EventQueue.invokeLater(new Runnable()
        {
            public void run()
            {
                new FenAppChart().setVisible(true);
            }
        });
    }
}

```

Ce qui donne :



A nouveau, l'objet chart utilise une instance d'une classe dérivée de la classe abstraite **Plot**, ici **CategoryPlot**, qui gère la représentation des données et leur référentiel avec des objets instances de **CategoryAxis** et **NumberAxis**, sans oublier un **BarRenderer**. Leur personnalisation approfondie relève de l'examen approfondi de la documentation, ce qui est de peu d'intérêt ici. Contentons-nous de

- ♦ changer la couleur des rectangles : il s'agit bien clairement d'un problème de rendu (comme dans les composants Swing) et c'est donc l'objet Renderer associé au Plot qui peut se charger des changements de couleur : on l'obtient avec

```
public CategoryItemRenderer getRenderer()
```

et l'objet obtenu (pour nous, ce sera plus précisément un BarRenderer) possède la méthode :

```
void setSeriesPaint (int series, java.awt.Paint paint)
```

Pour rappel, l'interface **Paint** possède notamment comme classe d'implémentation, la classe **Color** - cela donne donc ici, après obtention du JFreeChart jfc :

```
...
CategoryPlot plot = jfc.getCategoryPlot();
plot.getRenderer().setSeriesPaint(0, new Color(128, 128, 128));
plot.getRenderer().setSeriesPaint(1, new Color(0, 255, 0));
...
```

changer les repères de l'axe vertical : à nouveau, l'objet Plot (ou plutôt CategoryPlot pour être exact) peut nous fournir un objet matérialisant l'axe :

```
public ValueAxis getRangeAxis()
```

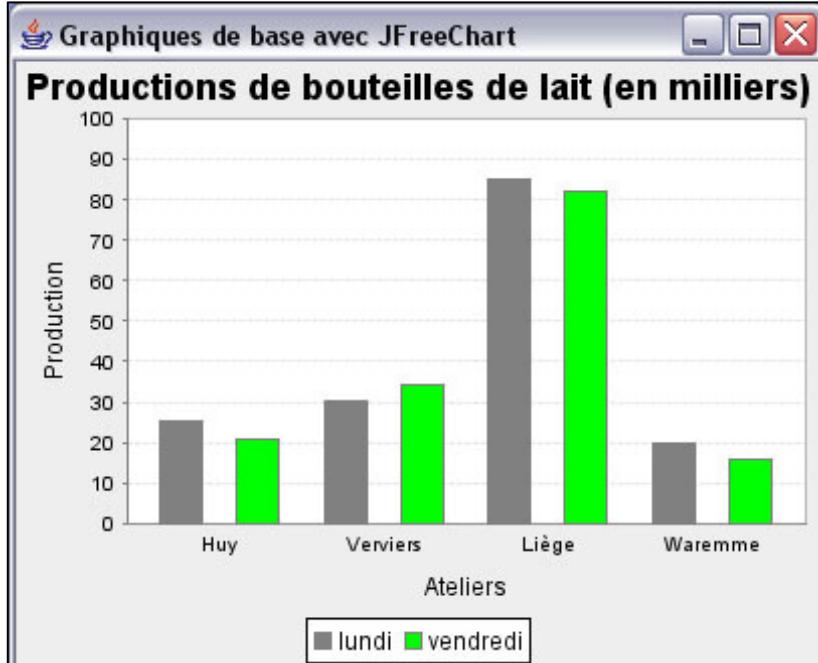
Pour nous, il s'agira plus précisément d'un NumberAxis, dont il n'y a plus qu'à changer le domaine et l'unité utilisée avec :

```
public void setRange (double lower, double upper)  
public void setTickUnit (NumberTickUnit unit)
```

un NumberTickUnit n'étant jamais qu'une encapsulation d'un double - cela donne donc ici, à la suite des instructions précédentes :

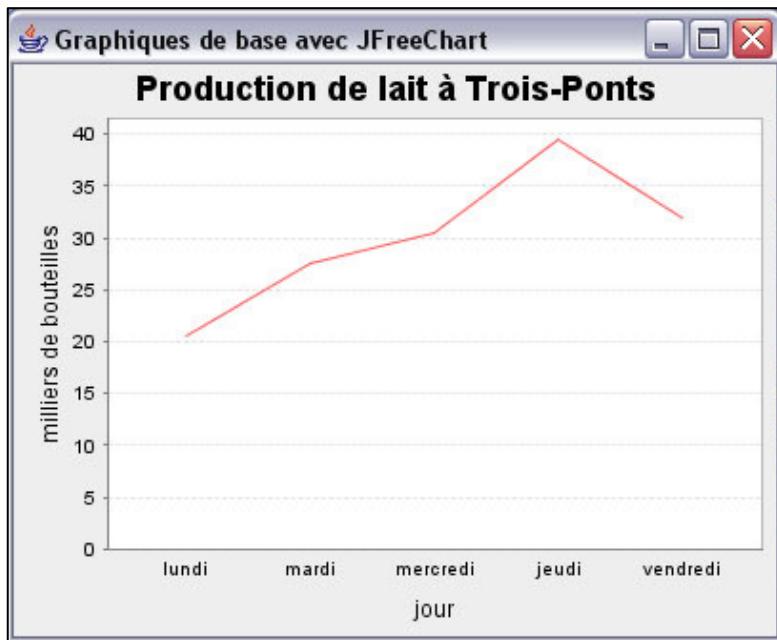
```
...  
NumberAxis rangeAxis = (NumberAxis) plot.getRangeAxis();  
rangeAxis.setRange(0.0, 100.0);  
rangeAxis.setTickUnit(new NumberTickUnit(10.0));  
....
```

L'exécution donnera :



5.2 Les graphiques linéaires d'évolution

Sans entrer dans les détails, signalons l'existence des graphiques linéaires classiques dont le but est de matérialiser une évolution (les "lignes brisées" classiques). L'idée est pas exemple de réaliser le graphique suivant :



Le dataset est analogue à celui des histogrammes. Mais on utilise cette fois la factory :

```
public static JFreeChart createLineChart(
    String title,
    String categoryAxisLabel,
    String valueAxisLabel,
    CategoryDataset dataset,
    PlotOrientation orientation,
    boolean legend,
    boolean tooltips,
    boolean urls)
```

Point n'est sans doute trop besoin d'insister :

FenAppChart.java (4)

```
...
private void showEvolution()
{
    String[] jours = {"lundi", "mardi", "mercredi", "jeudi", "vendredi"};
    double[] valProduction = { 20.5, 27.6, 30.4, 39.5, 31.9 };

    DefaultCategoryDataset ds = new DefaultCategoryDataset();
    for (int i=0; i<jours.length; i++)
        ds.addValue(valProduction[i], "Bouteilles", jours[i]);

    JFreeChart jfc = ChartFactory.createLineChart
        ("Production de lait à Trois-Ponts", "jour", "milliers de bouteilles",
         ds, PlotOrientation.VERTICAL, false, true, false);
    ChartPanel cp = new ChartPanel(jfc);
    this.getContentPane().add(cp);
}
```

6. Les graphiques de statistique à deux dimensions

6.1 Le nuage de points

Pour retrouver le monde de la statistique à deux dimensions, considérons par exemple le problème suivant (voir "Inférence statistique", chapitre VI – du même auteur). Le psychologue attaché à une entreprise soupçonne qu'il existe une corrélation entre la perception visuelle et la dextérité manuelle des ouvriers travaillant sur une chaîne d'assemblage. Il va donc vérifier cette assertion en faisant passer (durant les heures de travail) un test de réponse à un stimulus visuel (soit **X** la VA correspondante) et un autre test de dextérité manuelle (soit **Y** la VA correspondante) à un échantillon de 15 ouvriers choisis au hasard (on peut donc même y trouver un délégué syndical ...). Les résultats donnent 15 couples de valeurs (résultat perception visuelle, résultat dextérité manuelle) :

xi	yi			
3,8	68	3,5	74	
3,6	72	3,8	59	
3,4	86	4,1	55	
3,5	78	3,7	64	
3,9	64	3,6	73	
4,2	61	3,8	62	
3,7	66	3,4	75	
		3,5	78	

La construction du nuage de points correspondant au problème à visualiser est très similaire aux précédents :

- ◆ on crée tout d'abord un Dataset approprié à la situation soit un **XYDataset**. Cet interface dérivé de Dataset est implémenté notamment par la classe **XYSeriesCollection** (qui en fait dérive de la classe abstraite **AbstractIntervalXYDataset** qui implémente l'interface **IntervalXYDataset** dérivé de **XYDataset** – ouf !). Cette classe est instanciée le plus souvent avec un constructeur par défaut et enregistre des séries statistiques à deux dimensions (donc des listes de couples (x,y)) au moyen de sa méthode :

```
public void addSeries(XYSeries series)
```

- ◆ il faut donc créer le (ou les) série(s) qui sont des instances de la classe **XYSeries**. Le constructeur usuel est

```
public XYSeries (Comparable key)
```

- l'idée est que l'on peut ainsi retrouver chaque série par une clé – usuellement, un simple nom suffit. Les valeurs peuvent être dupliquées (un constructeur polymorphe permet d'interdire cela). On place des couples dans la série par la méthode :

```
public void add (double x, double y)
```

On s'en doute un peu, l'ajout d'un couple provoque l'envoi d'un **SeriesChangeEvent** à tous les listeners intéressés (ce qui est le cas du Dataset par l'intermédiaire de sa classe mère **AbstractIntervalXYDataset**).

- ♦ on peut enfin appeler la méthode factory de **ChartFactory** qui créera le nuage de points à partir du Dataset :

```
public static JFreeChart createScatterPlot (
    java.lang.String title,
    java.lang.String xAxisLabel,
    java.lang.String yAxisLabel,
    XYDataset dataset,
    PlotOrientation orientation,
    boolean legend,
    boolean tooltips,
    boolean urls)
```

et, comme d'habitude, il n'y a plus qu'à créer le ChartPanel et à le placer sur le ContentPane.

En résumé, cela donnera :

FenAppChart.java (5)

```
...
private void showScatterPlot()
{
    double couples[][] = { {3.8,68.0}, {3.6,72.0}, {3.4,86.0}, {3.5,78.0}, {3.9,64.0},
        {4.2,61.0}, {3.7,66.0}, {3.5,74.0}, {3.8,59.0}, {4.1,55.0},
        {3.7,64.0}, {3.6,73.0}, {3.8,62.0}, {3.4,75.0}, {3.5,78.0} };

    XYSeries serieObs = new XYSeries("Relations vision-manipulation");

    for (int i=0; i<15; i++)
        serieObs.add(couples[i][0],couples[i][1]);

    XYSeriesCollection ds = new XYSeriesCollection();
    ds.addSeries(serieObs);

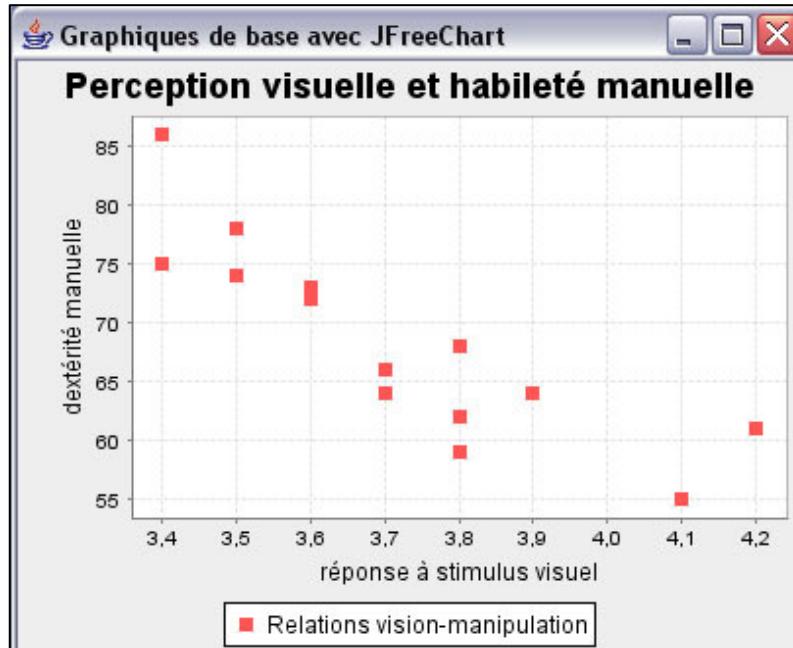
    JFreeChart jfc = ChartFactory.createScatterPlot(
        "Perception visuelle et habileté manuelle",
        "réponse à stimulus visuel", "dextérité manuelle",
        ds,
        PlotOrientation.VERTICAL,
        true, true, false );

    ChartPanel cp = new ChartPanel(jfc);

    this.getContentPane().add(cp);

}
```

Résultat :



Il est à soupçonner que nous soyons devant un phénomène de corrélation négative ...

6.2 Les paramètres de régression et de corrélation

La librairie fournit une classe utilitaire nommée **Statistics** qui possède un certain nombre de méthodes statiques permettant les calculs de moyenne et écart type pour une série statistique à une dimension, et des paramètres de régression et corrélation pour les séries statistiques à deux dimensions. Les séries considérées sont représentées par des tableaux d'objets **Number**, classe abstraite (du package `java.lang`) dont l'un des descendants est **Double** qui encapsule un réel en double précision. Les méthodes en question sont :

- ◆ public static double **calculateMean**(`java.lang.Number[]` values)
- ◆ public static double **getStdDev**(`java.lang.Number[]` data)

calculent respectivement la moyenne (m) et l'écart-type (s);

- ◆ public static double[] **getLinearFit**(`java.lang.Number[]` xData, `java.lang.Number[]` yData)

fournit dans le tableau résultat respectivement l'ordonnée à l'origine (b) et la pente (a) de la droite de régression ($y=ax+b$) calculée par une règle des moindres carrés;

- ◆ public static double **getCorrelation**(`java.lang.Number[]` data1, `java.lang.Number[]` data2)

fournit le coefficient de corrélation linéaire (r).

Nous pouvons donc ajouter à notre méthode dédiée aux nuages de points le code de calcul de ces paramètres. Une fois muni des valeurs de a et b , nous pouvons calculer les valeurs estimées (y_e) et en faire une deuxième série que nous ajouterons à notre Dataset – la magie des events-listeners fera le reste ...

FenAppChart.java (6)

```

...
private void showScatterPlot()
{
    ... // nuage de points

    Number[] x = new Double[15];
    Number[] y = new Double[15];
    for (int i=0; i<15; i++)
    {
        x[i] = couples[i][0];
        y[i] = couples[i][1];
    }
    double[] droiteRegression = Statistics.getLinearFit(x, y);
    double a = droiteRegression[1]; // et pas [0] !
    double b = droiteRegression[0];

    System.out.println("Droite de régression : y = " + a + "x + " + b);
    System.out.println("Droite de régression : y = " + viewDec(a,2)
        + "x " + (b<0?"":"+" ) + viewDec(b,2));

    double r = Statistics.getCorrelation(x,y);
    System.out.println("Coefficient de corrélation : r = " + r);
    System.out.println("Coefficient de corrélation : r = " + viewDec(r,2));

    double mx = Statistics.calculateMean(x);
    double sx = Statistics.getStdDev(x);
    System.out.println("VA X : m = " + mx + " et s = " + sx);
    System.out.println("VA X : m = " + viewDec(mx,2) + " et s = " + viewDec(sx,2));

    double my = Statistics.calculateMean(y);
    double sy = Statistics.getStdDev(y);
    System.out.println("VA Y : m = " + my + " et s = " + sy);
    System.out.println("VA Y : m = " + viewDec(my,2) + " et s = " + viewDec(sy,2));

    // Droite de régression
    XYSeries serieReg = new XYSeries("Droite de régression");
    for (double xe = 3.4; xe<= 4.2; xe+=0.1)
    {
        double ye = a*xe + b;
        System.out.println(xe + "," + ye );
        serieReg.add(xe,ye);
    }
    ds.addSeries(serieReg);
}

public String viewDec (double x, int nd)
{
    NumberFormat nf = NumberFormat.getInstance();

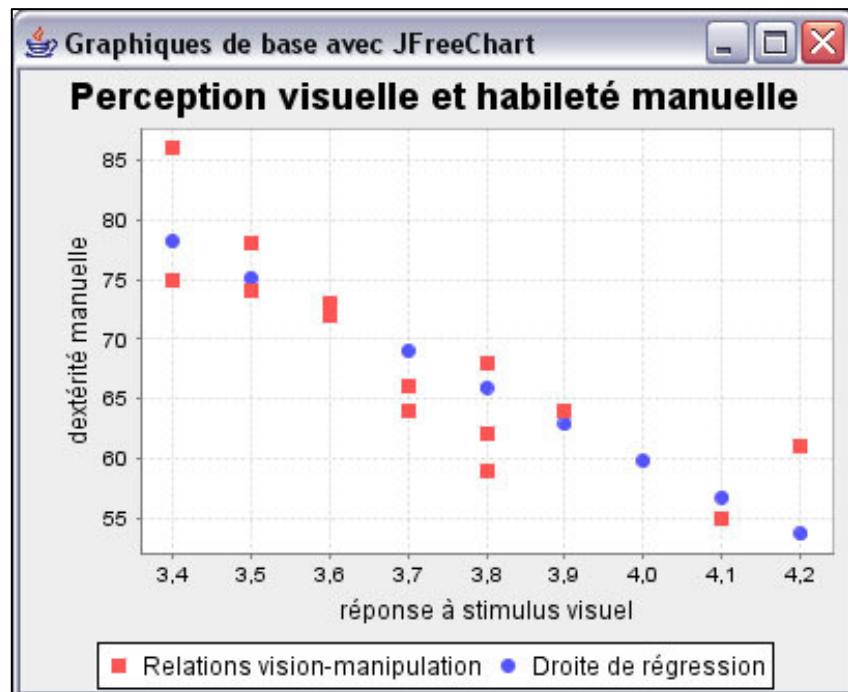
```

```
//System.out.println("Classe instanciée = " + nf.getClass().getName());
nf.setMinimumFractionDigits(nd);
nf.setMaximumFractionDigits(nd);
String str = nf.format(x);
return str;
}
...
```

Sur la console, cela donne :

Droite de régression : $y = -30.750000000000544x + 182.77500000000202$
 Droite de régression : **y = -30,75x +182,78**
 Coefficient de corrélation : $r = -0.8671437999440119$
 Coefficient de corrélation : **r = -0,87**
 VA X : m = 3.7 et s = 0.2390457218668787
 VA X : m = 3,70 et s = 0,24
 VA Y : m = 69.0 et s = 8.476859256655313
 VA Y : m = 69,00 et s = 8,48

et du point de vue graphique :



On remarquera l'opération cosmétique visant à limiter l'affichage des réels à deux décimales. Pour cela, nous utilisons la classe **NumberFormat** (du package `java.text`) ou plutôt, puisqu'elle est abstraite, sa classe dérivée **DecimalFormat**. En fait, nous ignorons cela car nous nous procurons l'objet au moyen de la factory :

```
public static final NumberFormat getInstance()
```

L'objet ainsi obtenu est capable de nous renvoyer une chaîne de caractères représentant le nombre réel formaté selon nos désirs, ceci au moyen de la méthode :

```
public final String format(double number)
```

après avoir, au préalable, configurer l'objet de formatage, par exemple avec les méthodes :

```
public void setMinimumFractionDigits (int newValue)  
public void setMaximumFractionDigits (int newValue)
```

7. Les séries chronologiques

7.1 Un axe des X avec des temps

Nous avons déjà réalisé un graphique d'évolution, c'est-à-dire fournissant un diagramme linéaire dont les valeurs sur X étaient fait des données temporelles (des mois, des jours, des temps). En fait, il existe pour ce cas particulier de diagramme en XY

- ◆ une classe dédiée pour le Dataset associé : la classe **TimeSeriesCollection** (qui dérive de la classe abstraite **AbstractIntervalXYDataset** et est donc une sœur de **XYSeriesCollections**) : si les valeurs x de ce genre de séries sont bien des double, elles représentent en réalité le nombre de millisecondes écoulées depuis le 1^{er} janvier 1970 et sont donc susceptibles d'être transformées en objet Date. C'est le rôle d'un objet **DateAxis** de réaliser cette conversion (mais ceci est transparent dans une utilisation standard).
- ◆ une classe **TimeSeries** pour peupler l'instance de **TimeSeriesCollection** de séries de données chronologiques : son constructeur le plus utile

```
public TimeSeries(String name, java.lang.Class timePeriodClass)
```

réclame essentiellement le nom de la classe qui fournira les instances successives représentant les temps écoulés régulièrement selon la période fixée par la nature de cette classe (jour, mois, minute, ...). Toutes les classes prévues par la librairie sont sensées implémenter l'interface **TimePeriod** qui déclare les méthodes :

```
java.util.Date getStart()  
java.util.Date getEnd()
```

- en fait, elles dérivent de la classe abstraite **RegularTimePeriod** aux méthodes additionnelles de manipulations des millisecondes encapsulées et aussi pourvue de la déclaration de deux méthodes :

```
public abstract RegularTimePeriod previous()  
public abstract RegularTimePeriod next()
```

Les classes possibles sont bien normalement : Day, FixedMillisecond, Hour, Millisecond, Minute, Month, Quarter, Second, Week et Year.

- ◆ une factory dédiée :

```
public static JFreeChart createTimeSeriesChart (  
    java.lang.String title,  
    java.lang.String timeAxisLabel,  
    java.lang.String valueAxisLabel,  
    XYDataset dataset,
```

```
boolean legend,
boolean tooltips,
boolean urls)
```

En pratique, supposons souhaiter comparer les données de production de milliers de litres de lait de deux exploitations agricoles en 2006. On peut tout d'abord imaginer, dans un objectif de généricté future, une classe ***Productions*** destinée à contenir des données de production (un tableau de double) pour un type de période donnée (par exemple, par mois) - ce type de période est désigné par une variable membre statique de la classe **Calendar**. De plus, la période supérieure (donc l'année dans le cas de mois) sera également encapsulée. Cela donne :

Productions.java

```
package jfreechartconcepts.data;

import java.util.Calendar;
import jfreechartconcepts.data.InvalidDataProduction;

/**
 * @author Vilvens
 */

public class Productions
{
    private int TypePeriode;
    private double data[];
    private int PeriodeSuperieure;

    public Productions (int t, double d[], int ps) throws InvalidDataProduction
    {
        Calendar c = Calendar.getInstance();
        TypePeriode = t; data = d; PeriodeSuperieure = ps;
        if (data.length != (c.getActualMaximum(TypePeriode)-
            c.getActualMinimum(TypePeriode)+1)) throw new InvalidDataProduction
            ("Nombre de données incohérent avec le type de période");
    }

    public int getTypePeriode() { return TypePeriode; }
    public void setTypePeriode(int TypePeriode) { this.TypePeriode = TypePeriode; }
    public double[] getData() { return data; }
    public void setData(double[] data) { this.data = data; }
    public int getPeriodeSuperieure() { return PeriodeSuperieure; }
    public void setPeriodeSuperieure(int PeriodeSuperieure)
        { this.PeriodeSuperieure = PeriodeSuperieure; }
}
```

avec une classe d'exception élémentaire utilisée si le nombre de données ne correspond pas au type de période choisi (par exemple, 11 données alors que l'on parle en mois d'une année) :

InvalidDataProduction.java

```
package jfreechartconcepts.data;

/**
 * @author Vilvens
 */

public class InvalidDataProduction extends Exception
{
    public InvalidDataProduction() {}

    public InvalidDataProduction(String msg) { super(msg); }
}
```

La réalisation du graphique statistique ne pose alors plus de problèmes :

FenAppChart.java (7)

```
...
private void showTimeSeries()
{
    double [][] ProductionsLait = {
        {65.23, 54.54, 59.71, 45.12,
         32.14, 32.19, 40.84, 46.21,
         47.67, 42.36, 45.65, 55.81 },

        {60.31, 57.54, 62.71, 49.12,
         30.14, 38.19, 44.84, 49.21,
         53.67, 54.36, 49.65, 56.81 }
    };

    String[] lieuxProductions = { "Trois-Ponts", "Stavelot"};
    int annee = 2006;

    Productions[] p = new Productions[2];
    try
    {
        for (int j=0; j<lieuxProductions.length; j++)
            p[j] = new Productions(Calendar.MONTH, ProductionsLait[j], annee);
    }
    catch (InvalidDataProduction e)
    {
        System.out.println("Oh oh ... " + e.getMessage());
    }

    Calendar c = Calendar.getInstance();

    TimeSeriesCollection ds = new TimeSeriesCollection();
    TimeSeries[] s = new TimeSeries[2];
```

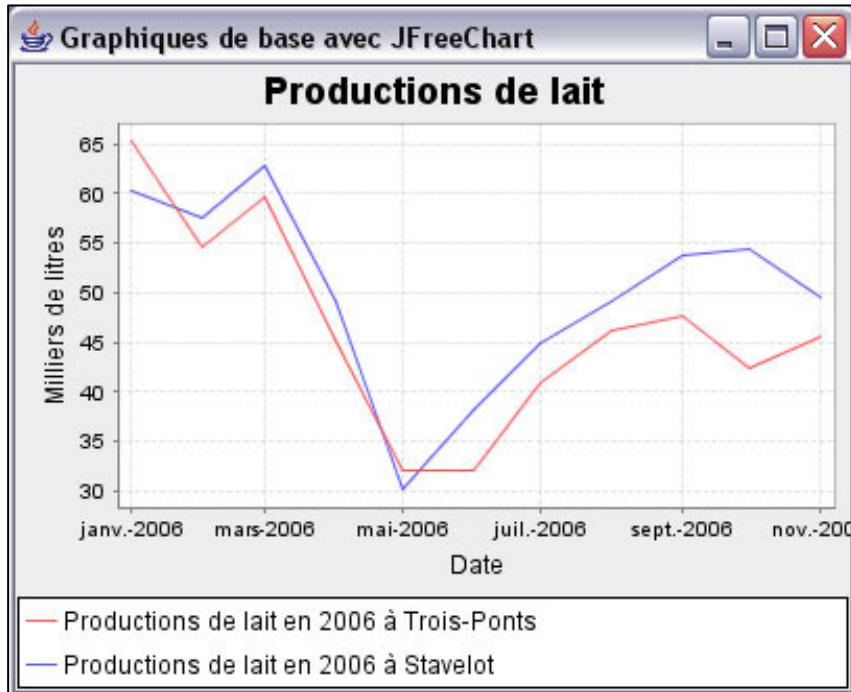
```

for (int j=0; j<lieuxProductions.length; j++)
{
    s[j] = new TimeSeries("Productions de lait en " + p[j].getPeriodeSuperieure() + " à "
        + lieuxProductions[j], Month.class);
    for (int i = 1; i<= c.getActualMaximum(p[j].getTypePeriode()); i++ )
    {
        s[j].add(new Month(i, p[j].getPeriodeSuperieure()), p[j].getData()[i-1]);
    }
    ds.addSeries(s[j]);
}

JFreeChart jfc = ChartFactory.createTimeSeriesChart(
    "Productions de lait",
    "Date", // x
    "Milliers de litres", // y
    ds,
    true, true, false
);
ChartPanel cp = new ChartPanel(jfc);
this.getContentPane().add(cp);
}
...

```

Ce qui nous donne :



7.2 Une graphique temporel à partir d'un base de données

Tout comme pour PieDataSet, il existe pour le XYDataSet un JDBCXYDataSet tout à fait analogue au JDBCPieDataSet. Remarquons cependant sa méthode :

public void setTimeSeries(boolean timeSeries)

permettant d'indiquer que les données considérées sont bien celles d'une série chronologique.

Pour reprendre un exemple similaire au précédent, nous pouvons créer une table contenant les données selon la règle que la 1^{ère} colonne correspond aux données en X (donc, ce sont les temps) tandis que les autres colonnes correspondent aux diverses séries. Cela pourrait donner :

```
mysql> create database GestionProduction;
Query OK, 1 row affected (0.11 sec)

mysql> use GestionProduction;
Database changed

mysql> grant all privileges on GestionProduction to 'CFSERVGESTION'@'localhost'
identified by 'ViveLeProfit' with grant option;
Query OK, 0 rows affected (0.27 sec)

mysql> CREATE TABLE productions (dateReleve DATE, dataTroisPonts DOUBLE,
dataStavelot DOUBLE);
Query OK, 0 rows affected (0.33 sec)

mysql> grant all privileges on productions to 'CFSERVGESTION'@'localhost' identi
fied by 'ViveLeProfit' with grant option;
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO productions VALUES ('2006-1-31', 65.20, 60.31);
Query OK, 1 row affected (0.06 sec)
mysql> INSERT INTO productions VALUES ('2006-2-28', 65.23, 60.28);
Query OK, 1 row affected (0.01 sec)
mysql> INSERT INTO productions VALUES ('2006-3-31', 54.54, 57.54);
Query OK, 1 row affected (0.03 sec)
mysql> INSERT INTO productions VALUES ('2006-4-30', 59.71, 62.71);
Query OK, 1 row affected (0.03 sec)

mysql> commit;
Query OK, 0 rows affected (0.00 sec)

mysql> select * from productions;
+-----+-----+-----+
| dateReleve | dataTroisPonts | dataStavelot |
+-----+-----+-----+
| 2006-01-31 |       65.2 |      60.31 |
| 2006-02-28 |       65.23 |     60.28 |
| 2006-03-31 |       54.54 |     57.54 |
| 2006-04-30 |       59.71 |     62.71 |
+-----+-----+-----+
4 rows in set (0.00 sec)
```

La méthode de construction du graphique est évidemment un héritage multiple ;-) de celle du graphique chronologique et de celle du graphique en camembert à partir d'une base de données :

FenAppChart.java (8)

```

...
private void showTimeSeriesJdbc()
{
    JDBCXYDataset jds = null;

    String url = "jdbc:mysql://localhost:3306/GestionProduction";
    try
    {
        Class.forName("com.mysql.jdbc.Driver");
    }
    catch (ClassNotFoundException e)
    {
        System.err.println(e.getMessage());
    }
    System.out.println("Driver MySQL (com) chargé");

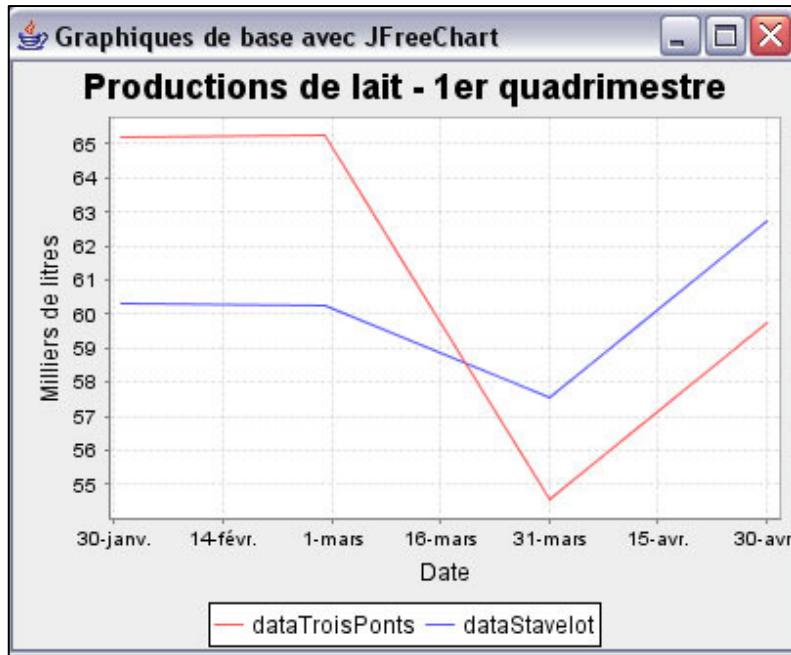
    Connection con = null;
    try
    {
        con = (Connection) DriverManager.getConnection(url, "CFSERVGESTION",
            "ViveLeProfit");
        System.out.println("Connexion à la BDD GestionProduction réalisée");
        jds = new JDBCXYDataset(con);
        jds.setTimeSeries(true);
        String req = "SELECT * FROM productions;";
        jds.executeQuery(req);
        System.out.println("Dataset chargé");
        con.close();
    }
    catch (SQLException e)
    {
        System.err.println(e.getMessage());
    }
    catch (Exception e)
    {
        System.err.println(e.getMessage());
    }

    if (jds.isTimeSeries()) System.out.println("Série temporelle");
    else System.out.println("Série PAS temporelle");
    JFreeChart jfc = ChartFactory.createTimeSeriesChart(
        "Productions de lait - 1er quadrimestre",
        "Date", // x
        "Milliers de litres", // y
        jds,
        true, true, false
    );
    ChartPanel cp = new ChartPanel(jfc);
    this.getContentPane().add(cp);

}
...

```

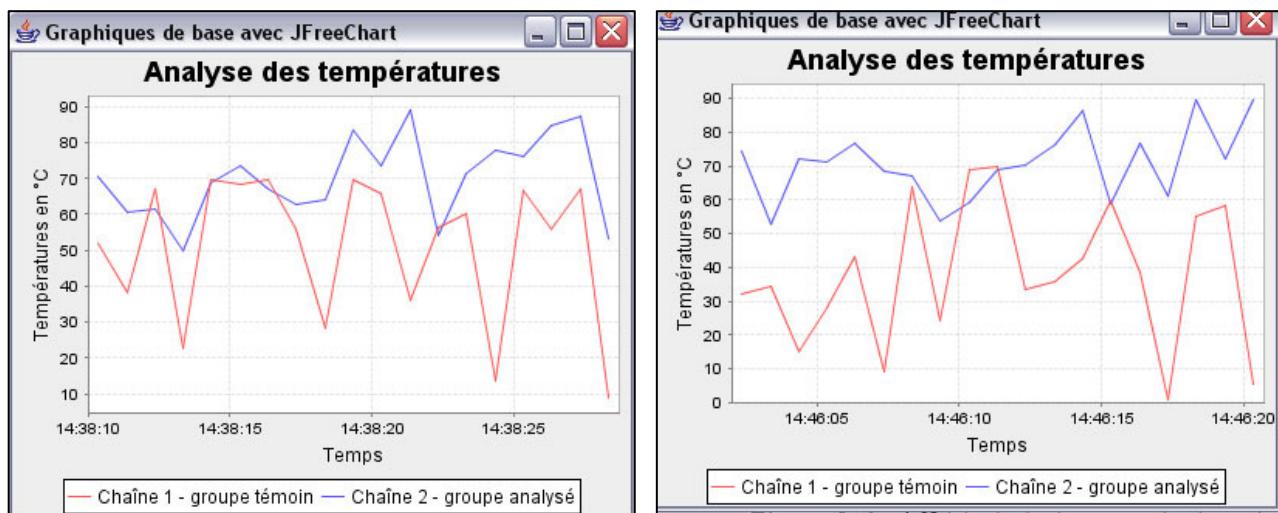
Le résultat cache cependant une petite surprise :



Vous avez vu laquelle ;-) ?

8. Les graphiques dynamiques

Une application pratique simple à réaliser consiste à visualiser un phénomène sur un graphique qui tient compte de nouvelles données au fur et à mesure qu'elles sont disponibles. Typiquement il s'agit par exemple de la surveillance de chaînes de fabrications. Non seulement de nouvelles données sont ajoutées, mais les plus anciennes sont éliminées afin de ne pas surcharger le graphique. On vise donc un résultat du type suivant, qui compare des mesures de températures relevées sur deux chaînes similaires, la première utilisant un groupe témoin :



Nous utiliserons pour cela un objet **Timer** (du package javax.swing) qui émet à intervalles réguliers un événement ActionEvent. Ici, il va modifier périodiquement les deux datasets utilisés. Son constructeur

```
public Timer(int delay, ActionListener listener)
```

réclame simplement sa période (en millisecondes) et, éventuellement, l'objet listener qui va réagir à l'émission de l'événement. On aura bien compris qu'il s'agit d'un thread dédié, que l'on fait démarrer avec sa méthode :

```
public void start()
```

L'addActionListener sera ici l'application elle-même et sa méthode actionPerformed() ajoutera une nouvelle donnée à chaque série de données – ces nouvelles données devraient provenir, par exemple, d'une socket ou d'une ligne série, mais nous les simulerons ici à coup de générateur de nombres aléatoires. Petite subtilité : un compteur de mesures observées est nécessaire afin que, parvenu à une certaine quantité de données, les premières soient éliminées des datasets au moyen de la méthode des TimeSeries ::

```
public void delete(int start, int end)
```

Sans trop insister, on peut donc écrire :

FenAppChart.java (9)

```
/*
 * FenAppChart.java
 */

package jfreechartconcepts;

import java.awt.Color;
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.Timer;
import org.jfree.chart.ChartFactory;
import org.jfree.chart.ChartPanel;
...
/***
 * @author Vilvens
 */

public class FenAppChart extends javax.swing.JFrame implements ActionListener
{
    private TimeSeries chaine1, chaine2;
    private int cptObs;
    //private DefaultPieDataset ds;
```

```

public FenAppChart()
{
    initComponents();
    getContentPane().setLayout(new GridLayout(1,1));
    cptObs = 0;

/*
    showPieChart(); showPieChartJdbc(); showHistogram(); showEvolution();
    showScatterPlot(); showTimeSeries(); showTimeSeriesJdbc();
*/
    showTimeSeriesEvolution();
}

private void initComponents() { ... }

private void showTimeSeriesEvolution()
{
    chaine1 = new TimeSeries("Chaîne 1 - groupe témoin", Millisecond.class);
    chaine2 = new TimeSeries("Chaîne 2 - groupe analysé", Millisecond.class);
    TimeSeriesCollection ds = new TimeSeriesCollection();
    ds.addSeries(chaine1);
    ds.addSeries(chaine2);

    JFreeChart jfc = ChartFactory.createTimeSeriesChart(
        "Analyse des températures",
        "Temps", // x
        "Températures en °C", // y
        ds,
        true, true, false
    );
    ChartPanel cp = new ChartPanel(jfc);
    this.getContentPane().add(cp);

    Timer t =new Timer (1000, this);
    t.start();
}

public void actionPerformed(ActionEvent e)
{
    chaine1.add(new Millisecond(), Math.abs(70*Math.sin(aleat(60.0,80.0)))); 
    chaine2.add(new Millisecond(), aleat(50.0,90.0));
    cptObs++;
    if (cptObs >= 20) { chaine1.delete(0,0) ; chaine2.delete(0,0) ; }
}

private double aleat (double bi, double bs)
{
    return bi + Math.random()*(bs-bi);
}
...
}

```

9. Les graphiques statistiques dans les applications Web

9.1 Une applet d'affichage

On pourrait évidemment imaginer d'afficher des graphiques statistiques au sein d'une applet : il suffirait de placer les instructions correspondantes dans la méthode init() de l'applet. Cela fonctionne (à condition que le browser utilisé possède la plugin Java nécessaire) mais avec le terrible handicap du transfert des jars de librairie nécessaires en plus du jar de l'applet elle-même. Il faut en effet utiliser un tag <APPLET> du genre lourd :

```
<APPLET ARCHIVE="appletchart.jar, jfreechart-1.0.6.jar, jcommon-1.0.10.jar"
CODE="charts.appletNuagePoints" width=640 height=260
ALT="Si vous voyez ce texte, il y a un problème avec votre browser ...">
</APPLET>
```

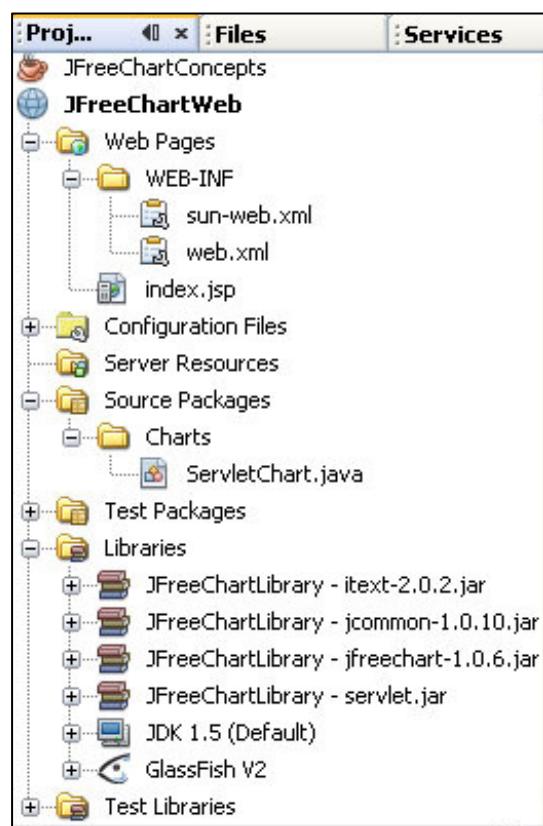
De toute manière, les restrictions de sécurité d'une applet font que, dans un contexte où l'accès aux données est toujours nécessaire, la solution d'une ou plusieurs servlets doit être préférée.

9.2 Une servlet d'affichage

Ce qui vient immédiatement à l'esprit est évidemment de faire générer le graphique par une servlet qui, du côté serveur, utilisera la librairie JFreeChart. De fait, il n'y a pas de difficulté réelle à travailler ainsi puisqu'il existe une classe **ChartUtilities** (package org.jfree.chart) qui possède les méthodes :

```
public static void writeChartAsJPEG(java.io.OutputStream out, JFreeChart chart, int width,
int height) throws java.io.IOException
public static void writeChartAsPNG(java.io.OutputStream out, JFreeChart chart, int width,
int height) throws java.io.IOException
```

à la fonction bien claire. On conçoit sans peine qu'il suffit de passer comme premier paramètre le flux de sortie de la servlet pour que le protocole HTTP apporte au client Web le graphique souhaité. Et de fait, si nous créons une application Web destinée à tourner sur le serveur GlassFish intégré et à laquelle nous avons attaché la librairie JFreeChart :



nous pouvons écrire une servlet élémentaire :

ServletChart

```
package Charts;

import java.io.*;

import javax.servlet.*;
import javax.servlet.http.*;
import org.jfree.chart.ChartFactory;
import org.jfree.chart.ChartUtilities;
import org.jfree.chart.JFreeChart;
import org.jfree.chart.plot.PlotOrientation;
import org.jfree.data.xy.XYSeries;
import org.jfree.data.xy.XYSeriesCollection;

/**
 * @author Vilvens
 */

public class ServletChart extends HttpServlet {

    protected void processRequest(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
    {
        response.setContentType("image/jpg");
        OutputStream out = response.getOutputStream();
        try
        {
            showScatterPlot (out);
        }
        finally { 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 fournissant un graphique en nuage de points"; }

    synchronized private void showScatterPlot (OutputStream os) throws IOException
    {
        double couples[][] = { {3.8,68.0}, {3.6,72.0}, {3.4,86.0}, {3.5,78.0}, {3.9,64.0},
            {4.2,61.0}, {3.7,66.0}, {3.5,74.0}, {3.8,59.0}, {4.1,55.0},
            {3.7,64.0}, {3.6,73.0}, {3.8,62.0}, {3.4,75.0}, {3.5,78.0} };
    }
}
```

```
XYSeries serieObs = new XYSeries("Relations vision-manipulation");

for (int i=0; i<15; i++)
    serieObs.add(couples[i][0],couples[i][1]);

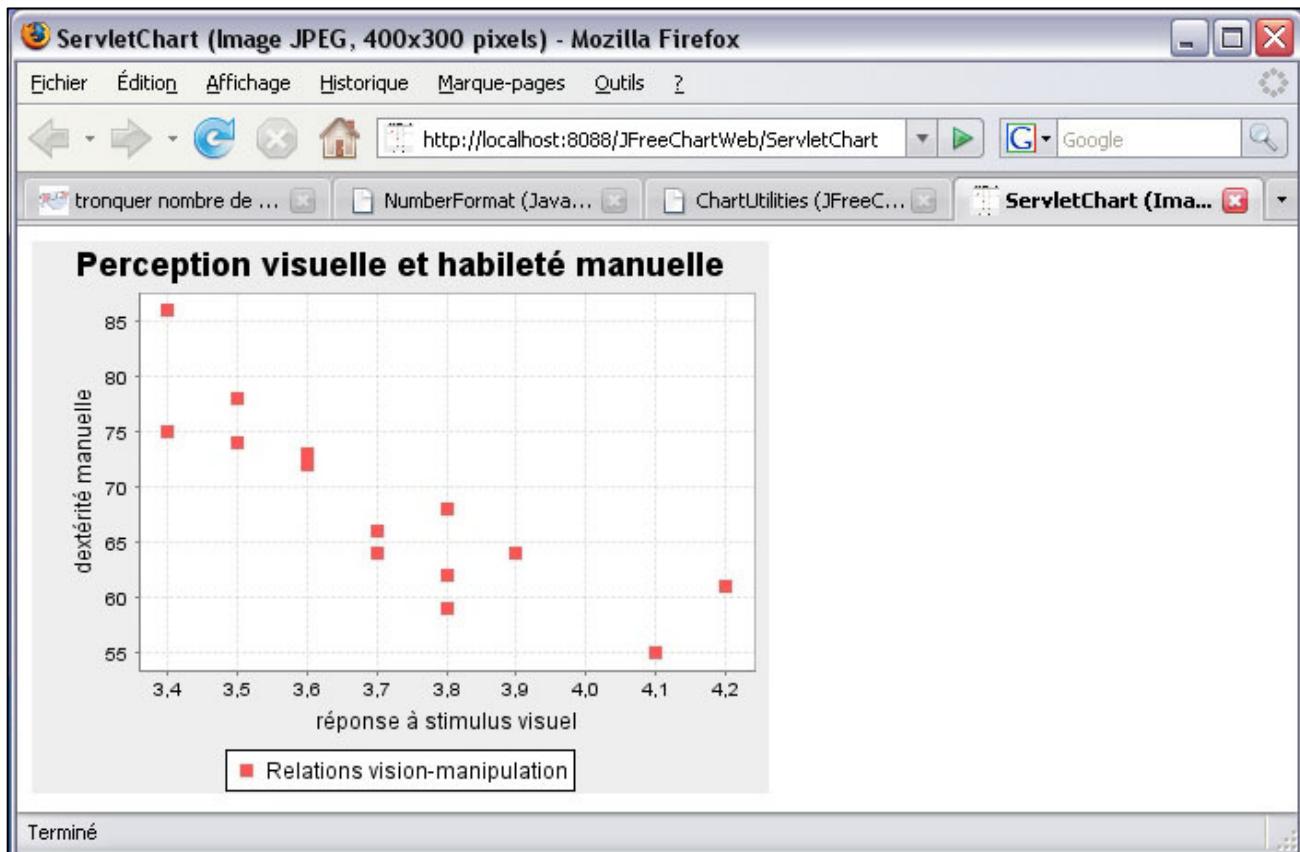
XYSeriesCollection ds = new XYSeriesCollection();
ds.addSeries(serieObs);

JFreeChart jfc = ChartFactory.createScatterPlot(
    "Perception visuelle et habileté manuelle",
    "réponse à stimulus visuel", "dextérité manuelle",
    ds,
    PlotOrientation.VERTICAL,
    true, true, false );

ChartUtilities.writeChartAsJPEG (os, jfc, 400, 300);
}

}
```

Le résultat est simple mais foudroyant :



9.3 Un servlet fournit une image

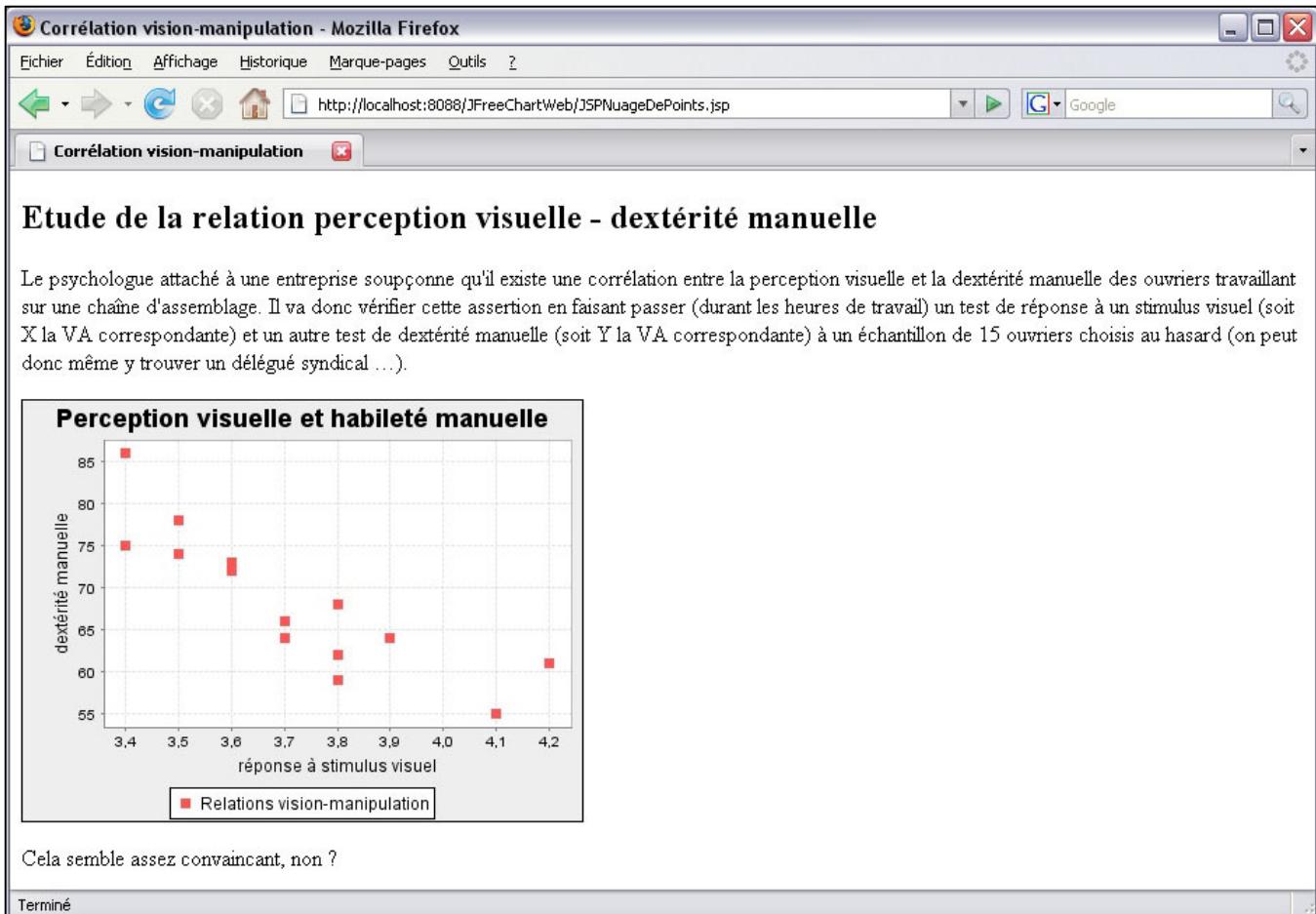
Evidemment, ce n'est pas très "décoré" comme résultat : on va donc plutôt considérer que le client charge un JSP dans lequel un simple tag demandera le graphique à la servlet :

JSPNageDePoints.jsp

```
<%--  
 Document : JSPNageDePoints  
 Author : Vilvens  
--%>  
  
<%@page contentType="text/html" 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>Corrélation vision-manipulation</title>  
 </head>  
 <body>  
   <h2>Etude de la relation perception visuelle - dextérité manuelle</h2>  
   Le psychologue attaché à une entreprise soupçonne qu'il existe une corrélation entre la  
   perception visuelle et la dextérité manuelle des ouvriers travaillant sur une chaîne d'assemblage. Il  
   va donc vérifier cette assertion en faisant passer (durant les heures de travail) un test de réponse à  
   un stimulus visuel (soit X la VA correspondante) et un autre test de dextérité manuelle (soit Y la  
   VA correspondante) à un échantillon de 15 ouvriers choisis au hasard (on peut donc même y  
   trouver un délégué syndical ...).  
   <p></p> <IMG SRC="ServletChart" BORDER=1 WIDTH=400 HEIGHT=300 />  
   <p><p></p>Cela semble assez convaincant, non ?  
 </body>  
</html>
```

Si la page index de l'application Web point sur le JSP, cela donnera, sans la moindre modification de la servlet :





9.4 Une servlet fournituse d'étude statistique

Bien sûr, disposer du nuage de points est un peu limitatif pour le client : il souhaitera probablement connaître la valeurs des coefficients a, b et r de la statistique à deux dimensions des données considérées. Pour que le JSP invoqué par le client puisse afficher ces paramètres, nous allons utiliser un bean assez prévisible, puisque ses propriétés utilisables dans le JSP par les tags <jsp:getProperty>, seront précisément les paramètres demandés sous formes de chaînes de caractères – les autres propriétés sont les séries en x et y, le nombre d'observations et les valeurs numériques de ces paramètres :

LinearRegCorrBean.java

```
package Charts;

import java.beans.*;
import java.io.Serializable;
import java.text.NumberFormat;
import org.jfree.data.statistics.Statistics;

/**
 * @author Vilvens
 */
```

```

public class LinearRegCorrBean extends Object implements Serializable
{
    private Number[] x,y;
    private double[] ParamReg;
    private double CorrCoeff;
    private int nbObs;

    private PropertyChangeSupport propertySupport;

    public LinearRegCorrBean()
    {
        nbObs = 15;
        double couples[][] = { {3.8,68.0}, {3.6,72.0}, {3.4,86.0}, {3.5,78.0}, {3.9,64.0},
            {4.2,61.0}, {3.7,66.0}, {3.5,74.0}, {3.8,59.0}, {4.1,55.0},
            {3.7,64.0}, {3.6,73.0}, {3.8,62.0}, {3.4,75.0}, {3.5,78.0} };

        propertySupport = new PropertyChangeSupport(this);
        x = new Double[nbObs]; y = new Double[nbObs];
        for (int i=0; i<nbObs; i++)
        {
            x[i] = couples[i][0]; y[i] = couples[i][1];
        }
        ParamReg = this.getParamReg();
        CorrCoeff = this.getCorrCoeff();
    }

    public Number[] getX() { return x; }
    public void setX(Number[] x) { this.x = x; }
    public Number[] getY() { return y; }
    public void setY(Number[] y) { this.y = y; }

    public double[] getParamReg() { return Statistics.getLinearFit(x, y); }
    public String getAParamReg() { return viewDec(ParamReg[1],2); }
    public String getBParamReg() { return viewDec(ParamReg[0],2); }

    public double getCorrCoeff() { return Statistics.getCorrelation(x,y); }
    public String getRCorrCoeff() { return viewDec(CorrCoeff,2); }

    public int getNbObs() { return nbObs; }
    public void setNbObs(int nbObs) { this.nbObs = nbObs; }

    public String viewDec (double x, int nd)
    {
        NumberFormat nf = NumberFormat.getInstance();
        System.out.println("Classe instanciée = " + nf.getClass().getName());
        nf.setMinimumFractionDigits(nd); nf.setMaximumFractionDigits(nd);
        String str = nf.format(x);
        return str;
    }
}

```

Bien sûr, un tel bean n'est pas un chef d'œuvre ;-) : il faudrait pouvoir le configurer avec des données dynamiques fournies par une base de données – autrement dit, il faudrait un MVC orthodoxe. To do ! Pour l'heure, contentons-nous d'exploiter ce bean dans notre JSP :

JSPNuageDePoints.jsp

```
<%--  
 Document : JSPNuageDePoints  
 Created on : 04-sept.-2008, 8:48:41  
 Author : Vilvens  
--%>  
  
<%@page contentType="text/html" pageEncoding="UTF-8"%>  
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"  
 "http://www.w3.org/TR/html4/loose.dtd">  
<jsp:useBean id="Statistique2D" scope="page" class="Charts.LinearRegCorrBean" />  
<html>  
 <head>  
   <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">  
   <title>Corrélation vision-manipulation</title>  
 </head>  
 <body>  
   <h2>Etude de la relation perception visuelle - dextérité manuelle</h2>  
   Le psychologue attaché à une entreprise soupçonne qu'il existe une corrélation entre la  
   perception visuelle et la dextérité manuelle des ouvriers travaillant sur une chaîne  
   d'assemblage. Il va donc vérifier cette assertion en faisant passer (durant les heures de travail)  
   un test de réponse à un stimulus visuel (soit X la VA correspondante) et un autre test de  
   dextérité manuelle (soit Y la VA correspondante) à un échantillon de 15 ouvriers choisis au  
   hasard (on peut donc même y trouver un délégué syndical ...).  
   <p></p> <IMG SRC="ServletChart" BORDER=1 WIDTH=320 HEIGHT=240 />  
   <p><p></p>Cela semble assez convaincant, non ?  
   Paramètre statistiques :  
   <p><h3>Droite de régression linéaire :</h3></p>  
   a = <jsp:getProperty name="Statistique2D" property="AParamReg" />  
   <p></p>b = <jsp:getProperty name="Statistique2D" property="BParamReg" />  
   <p><h3>Coefficient de corrélation :</h3></p>  
   r = <jsp:getProperty name="Statistique2D" property="RCorrCoeff" />  
 </body>  
</html>
```

Le résultat est sans surprise :

Corrélation vision-manipulation - Mozilla Firefox

Fichier Édition Affichage Historique Marque-pages Outils ?

AbstractIntervalYDataset (JFreeCha... Corrélation vision-manipulation

Etude de la relation perception visuelle - dextérité manuelle

Le psychologue attaché à une entreprise soupçonne qu'il existe une corrélation entre la perception visuelle et la dextérité manuelle des ouvriers travaillant sur une chaîne d'assemblage. Il va donc vérifier cette assertion en faisant passer (durant les heures de travail) un test de réponse à un stimulus visuel (soit X la VA correspondante) et un autre test de dextérité manuelle (soit Y la VA correspondante) à un échantillon de 15 ouvriers choisis au hasard (on peut donc même y trouver un délégué syndical ...).

Perception visuelle et habileté manuelle

Réponse à stimulus visual (X)	Dextérité manuelle (Y)
3.4	85
3.4	75
3.2	78
3.2	75
3.0	70
3.7	65
3.7	62
3.7	60
3.8	65
3.9	62
4.1	55
4.2	60

Cela semble assez convaincant, non ? Paramètre statistiques :

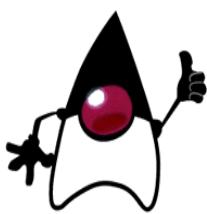
Droite de régression linéaire :

$a = -30,75$

$b = 182,78$

Coefficient de corrélation :

$r = -0,87$



à suivre ...

Connaître tout Java ? N'y pensez même pas : c'est probablement impossible ... Et, de toute manière, nous n'avons encore rien dit à propos des applications WEB ! Pas question cependant d'entreprendre une telle étude ici : ce sera pour le volume III ;-)

Annexe 1 : Les standards PKCS



Les **PKCS** (Public Key Cryptography Standards) sont un cadre de référence du domaine PKI publié par la société multinationale **RSA Data Security** (<http://www.rsa.com/>), spécialisée dans la sécurité cryptographique. Elle a développé, en concertation plus ou moins élaborée avec d'autres sociétés travaillent dans le domaine de la sécurité logicielle, un ensemble de *spécifications* portant sur différents aspects du domaine de la cryptographie à clé publique (chiffrements, certificats, périphériques cryptographiques, ...). Bien que RSA ne soit pas un organisme officiel de normalisation, ses spécifications (on devrait donc plutôt parler de "recommandations") sont devenues des *standards* de fait largement utilisés par le monde informatique. Ils sont numérotés de 1 à 15.

Un groupe de travail de l'IETF, le **PKIX** (Public-Key Infrastructure X.509) est chargé de définir dans des RFCs les normes officielles⁸. Certains des PKCS ont ainsi été redéfinis comme des RFC, devenant ainsi de véritables standards.

PKCS#x	version	nom	sujet (ce qui est défini)
PKCS#1 (avec PKCS#2 et 4 intégrés)	2.1	RSA Cryptography Standard	Cryptage RSA (formats, mécanismes, etc) – RFC 3447
PKCS#3	1.4	Diffie-Hellman Key Agreement Standard	Mécanisme d'obtention d'une clé de session par deux parties communicant par un canal non sécurisé et ne se connaissant pas au préalable.
PKCS#5	2.0	Password-based Encryption Standard	Mécanisme de création d'une clé cryptographique à partir d'un mot de passe (qui fait office de secret partagé) – RFC 2898
PKCS#6	1.5	Extended- Certificate Syntax Standard	Extensions aux spécifications de la 1 ^{ère} des certificats X509 – obsolète car intégrée dans la version 3 de X 509.
PKCS#7	1.5	Cryptographic Message Syntax Standard	Format des messages utilisés dans un contexte PKI, donc signées – à la base du format S/MIME – RFC 3852
PKCS#8	1.2	Private-Key Information Syntax Standard	Syntaxe des informations que l'on obtenir pour une clé privée (dans le but de l'utiliser à bon escient)
PKCS#9	2.0	Selected Attribute Types	Divers attributs utilisés dans les PKCS#6, 7, 8 et 10.
PKCS#10	1.7	Certification Request Standard	Format des CSR (demande de certificat à une autorité) – RFC 2986.

⁸ pour le texte des RFCs, voir <http://tools.ietf.org/rfc/index>

PKCS#11	2.20	Cryptographic Token Interface (Cryptoki)	API standard d'accès aux données cryptographiques mémorisées dans un dispositif hardware
PKCS#12	1.0	Personal Information Exchange Syntax Standard	Format de fichier permettant de mémoriser une clé privée accompagnée des certificats associés, le tout protégé par un mot de passe (entrée Key Entry d'un keystore)
PKCS#13	en cours de développement	Elliptic Curve Cryptography Standard	Cryptographie basée sur les courbes elliptiques (algorithmes RSA utilisant une algèbre basée sur les points associés par une telle courbe)
PKCS#14	en cours de développement	Pseudo-random Number Generation	Génération des nombres aléatoires
PKCS#15	1.1	Cryptographic Token Information Format Standard	Mécanisme d'identification des utilisateurs de périphériques cryptographiques indépendant de l'implémentation du cryptoki par l'application (PKCS #11).

Annexe 2 : La sérialisation et le serialVersionUID



Les hommes se distinguent par ce qu'ils montrent et se ressemblent par ce qu'ils cachent.

(P. Valéry, Suite)

En tant que langage abstrait, Java nous a habitués à ne pas devoir nous soucier des détails d'implémentation qui sont d'ailleurs d'habitude soigneusement encapsulés. Ainsi en est-il de la sérialisation des objets, que ce soit sur un fichier ou sur un flux réseau.

Il arrive cependant que la mécanique interne se rappelle à notre bon souvenir ...

1. Une exception inattendue

Supposons que nous envisagions de sérialiser, sur fichier ou sur tube TCP, les objets d'une modeste classe carteIdentite :

carteIdentite.java

```
package classics;

import java.io.Serializable;

public class carteIdentite implements Serializable
{
    protected String nom;
    protected String prenom;
    protected String numCId;

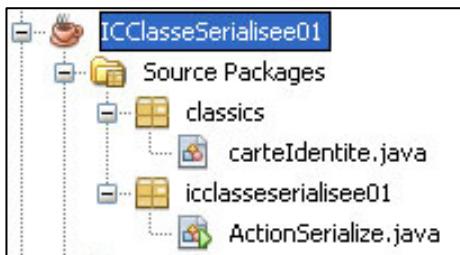
    public carteIdentite (String n, String p, String num)
    {
        this.setNom(n);this.setPrenom(p);this.setNumCId(num);
    }
    public String getPrenom() { return prenom; }
    final public void setPrenom(String prenom) { this.prenom = prenom; }
    public String getNumCId() { return numCId; }
    final public void setNumCId(String numCId) { this.numCId = numCId; }
    public String getNom() { return nom; }
    final public void setNom(String nom) { this.nom = nom; }
```

@Override

```
public String toString()
{
    return this.getNom() + " " + this.getPrenom() + "[" + this.getNumCId() + "]";
}
```

Rien d'extraordinaire, bien sûr. Supposons encore que l'on ait réalisé **avec succès** les tests préliminaires au sein d'un projet (au sein duquel on sérialise et désérialise), puis que l'on ait "splitté" ce projet initial en deux projets distincts (disons le "client" et "le serveur", le client voulant transmettre des objets au serveur). Cela pourrait donner :

a) **projet "client"**



ActionSerialize.java

```
package icclasseserialisee01;

import classics.carteIdentite;
import java.io.*;
import java.lang.reflect.Field;
import java.net.Socket;
import java.util.logging.Level;
import java.util.logging.Logger;

/**
 *
 * @author Vilvens
 */
public class ActionSerialize
{
    public static void main(String[] args)
    {
        carteIdentite ci = new carteIdentite("Vilvens", "Denys", "arglxx007");
        carteIdentite ci2 = new carteIdentite("Dugenou", "Félicien", "alest008");

        OutputStream os = null;
        try
        {
            if (args.length==0 || args[0].equals("FILE"))
                os = new FileOutputStream("identites.data");
            else if (args[0].equals("NETWORK"))
            {
                Socket s = new Socket ("localhost", 50000);
                os = s.getOutputStream();
            }
            else
            {
                System.out.println(
                    "Usage: java -jar ICClasseSerialisee01.jar [FILE|NETWORK]");
            }
        }
    }
}
```

```
        return;
    }
ObjectOutputStream oos = new ObjectOutputStream(os);
oos.writeObject(ci);oos.writeObject(ci2);
oos.close(); os.close();
if (args[0].equals("NETWORK")) return;
}
catch (IOException ex) {
    Logger.getLogger(ActionSerialize.class.getName()).log(Level.SEVERE, null, ex);
}
}
```

b) projet "serveur"



Action Deserialize.java

```
package iclasseserialisee02;

import classics.carteIdentite;
import java.io.*;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.logging.Level;
import java.util.logging.Logger;

/*
 * @author Vilvens
 */

public class ActionDeserialize
{
    public static void main(String[] args)
    {
        carteIdentite ci = null; carteIdentite ci2 = null;
        InputStream is = null;

        try
        {
            if (args.length==0 || args[0].equals("FILE"))
                is = new FileInputStream("identites.data");
        }
    }
}
```

```

else if (args[0].equals("NETWORK"))
{
    ServerSocket s = new ServerSocket(50000);
    Socket sc = s.accept();
    is = sc.getInputStream();
}
else
{
    System.out.println("Usage: java -jar ICClasseDeserialisee02.jar [FILE]")
    return;
}
ObjectInputStream ois = new ObjectInputStream(is);
ci = (carteIdentite) ois.readObject();
System.out.println("Objet lu " + ci.toString());
Class desc = ci.getClass(); System.out.println(desc.getCanonicalName());
ci2 = (carteIdentite) ois.readObject();
System.out.println("Objet lu " + ci2.toString());
ois.close();
}
catch (ClassNotFoundException ex)
{
    Logger.getLogger(ActionDeserialize.class.getName()).log(Level.SEVERE, null, ex);
}
catch (IOException ex) {
    Logger.getLogger(ActionDeserialize.class.getName()).log(Level.SEVERE, null, ex);
}
}

```

Bien sûr, selon que l'on sérialise en fichier ("FILE" sur la ligne de commande) ou sur pipe TCP ("NETWORK" sur la ligne de commande), le mode d'emploi diffère :

- ♦ en fichier : on exécute le premier projet, puis on copie le fichier de sérialisation dans le répertoire du second projet, qu'on exécute ensuite pour relire les objets;
 - ♦ en réseau (ici, en localhost) : on exécute le second projet (le "serveur"), puis le premier (le "client").

Mais, dans les deux cas, la deuxième exécution montre que l'interpréteur n'est pas content :

GRAVE: null

```
java.io.InvalidClassException: classics.carteIdentite; local class incompatible: stream  
classdesc serialVersionUID = 1, local class serialVersionUID = 7038822736417302289  
    at java.io.ObjectStreamClass.initNonProxy(ObjectStreamClass.java:562)  
    at java.io.ObjectInputStream.readNonProxyDesc(ObjectInputStream.java:1583)  
    at java.io.ObjectInputStream.readClassDesc(ObjectInputStream.java:1496)  
    at java.io.ObjectInputStream.readOrdinaryObject(ObjectInputStream.java:1732)  
    at java.io.ObjectInputStream.readObject0(ObjectInputStream.java:1329)  
    at java.io.ObjectInputStream.readObject(ObjectInputStream.java:351)  
    at iclasseserialisee02.ActionDeserialize.main(ActionDeserialize.java:46)
```

De quoi s'agit-il ? En fait, cette exception **InvalidClassException** hérite de **ObjectStreamException** et est lancée si la classe contient des données de type inconnu (mais ce n'est pas le cas ici), si un constructeur par défaut est inaccessible (il faudrait en avoir déclaré un qui soit privé – mais ce n'est pas le cas ici) ou encore parce que le numéro de version de sérialisation de la classe dont on souhaite récupérer des objets ne correspond pas au numéro de version trouvé dans le fichier : et c'est ce qui se passe ici ...

2. Le serialVersionUID

Nous découvrons donc que le mécanisme de sérialisation/désérialisation associé à toute classe implémentant **Serializable** un identifiant propre nommé le "serialVersionUID" : comme déjà dit, *il doit y avoir correspondance entre numéro de version de sérialisation de la classe dont on souhaite récupérer des objets et le numéro de version trouvé dans le flux* (fichier, réseau, ...).

Notre classe **carteIdentite** ne déclare pas explicitement un tel UID, si bien que **celui-ci est généré au moment de la sérialisation par le JRE**, en fonction du contenu de la classe (une spécification précise existe à ce sujet) mais aussi du contexte (notamment celui fourni par l'implémentation du compilateur Java utilisé et aussi par l'IDE éventuel). Dans notre cas de figure, les classes **carteIdentite** des deux projets vont se voir affectés des numéros de version différents, puisqu'elles seront compilées dans deux projets différents (qui dissimule donc un **CLASSPATH** différent) : on comprend donc bien la source du problème ...

On peut évidemment penser à orner notre classe d'un numéro de version que nous contrôlerons nous-mêmes : il suffit d'ajouter une variable membre statique de type long selon la syntaxe :

```
<public|protected|private> static final long serialVersionUID = <valeur entière>L;
```

On peut imaginer une valeur 1L pour débuter, en incrémentant ce numéro lorsqu'on ajoute de nouveaux membres (comme par exemple un champ transient). Nous allons donc ajouter à notre classe, dans les deux projets, la variable membre :

carteIdentite.java (2)

```
package classics;

import java.io.Serializable;

public class carteIdentite implements Serializable
{
    protected String nom;
    protected String prenom;
    protected String numCId;
    public static final long serialVersionUID = 1L;
    ...
}
```

Après recompilation des deux projets, on obtient une exécution correcte avec lecture des objets sérialisés :

| Objet lu Vilvens Denys[arglxx007]

classics.carteIdentite
Objet lu Dugenou Félicien[alest008]

3. Avec les jars, pas de problème

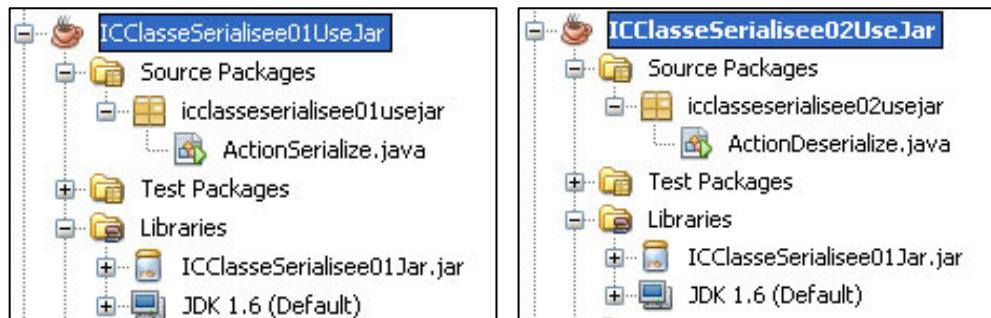
Bien que donc conseillé par les concepteurs de Java, on peut ressentir cette démarche de gestion d'UID comme bien lourde. Et, en fait, elle n'est pas forcément nécessaire : il suffit de

- ◆ placer la classe commune aux deux projets dans un jar :



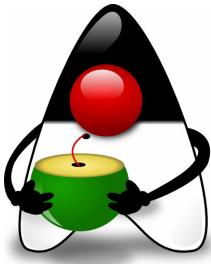
et cela sans la variable membre serialVersionUID;

- ◆ monter le jar dans les deux projets :



Evidemment, la classe étant cette fois unique en sa version, le problème disparaît de lui-même ;-)

Annexe 3 : Une introduction à JPA



Le temps n'existe que dans l'instant présent, un instant qui prend sa densité entre le regret de ce qui fut et le sourire de ce qui sera.

(Pierre-Jean Roland – utilisé dans une publicité pour Leffe)

1. Le couple Objets-Bases de données

Le développeur Java qui souhaite rendre persistants les objets de son développement se voit proposé à l'heure actuelle un certains nombre de techniques, plus généralement de "frameworks" (il faudra bien s'y faire, ce mot est entré dans le français technique). On peut ainsi citer :

- ◆ la **sérialisation** classique : elle est capable de sauver tout un graphe d'objets en relation mais, précisément, peine à sauver chaque groupe d'entités séparément;
- ◆ **JDBC** : qui propulse bien dans le monde bases de données, mais oblige le développeur à transformer lui-même les objets et leurs relations en tuples;
- ◆ les **frameworks ORM (Object-Relational Mapping)** : comme leur nom l'indique, ils assurent la correspondance objets-tuples de manière transparente; leur seul désavantage est d'utiliser chacun une API propriétaire, qui rendent évidemment les applications développées tributaires du "vendor";
- ◆ les **bases de données ODB (Object DataBases)** : autrement dit, c'est le SGBD qui va vers les objets; mais, à nouveau, une véritable normalisation se fait attendre, malgré les tentatives de l'ODMG (Object Database Management Group); de plus, le nombre d'outils d'analyse de telles bases est limité;
- ◆ les **entités des EJBs (Enterprise Java Beans)** : il s'agit de vues orientées objets des données persistantes; mais leurs limites en O.O. même et surtout leur complexité même les desservent, malgré la stricte standardisation dont le framework EJB a fait l'objet.

JPA (Java Persistence API Architecture) et **JDO** (Java Data Objects) se proposent de réunir les meilleurs éléments des diverses solutions évoquées ci-dessus :

- ◆ les entités sont créées comme de simples classes (les **POJOs** : Plain Old Java Objects);
- ◆ les techniques d'interrogations, de concurrence et de transactions sont disponibles, comme en JDBC;
- ◆ les techniques classiques de la POO sont disponibles (héritage, polymorphisme, etc);
- ◆ une spécification très stricte évite toute dépendance par rapport aux vendors.

JDO vise tous les types de SGBDs, ce qui le complique ses spécifications tandis JPA vise uniquement les SGBDs relationnels, ce qui simplifie la forme de l'API. On croit souvent que JPA est en fait un outil dédié aux EJB, et il est vrai que ces derniers en font usage. Cependant, on peut utiliser JPA dans des applications conventionnelles, comme nous allons le voir ...

2. L'architecture de JPA

L'élément de base de JPA est l'**entité** [entity] : il s'agit de la classe dont on souhaite rendre les instances persistantes. Elle sera annotée au moyen de l'annotation **@Entity**. Il lui est demandé de posséder un constructeur sans arguments (par défaut – encore que cette terminologie soit discutable). Le mécanisme de persistance proprement dit est assuré par un **EntityManager** (interface du package javax.persistence) : il assure la persistance d'un ensemble d'objets au moyen de sa méthode

`void persist(Object entity)`

et en permet la gestion (interrogation, manipulation) au moyen de méthodes comme

- ◆ `Query createNamedQuery(String name)`

où le paramètre est une requête en **JPQL** (Java Persistence Query Language) ou en SQL classique. On s'en doute, l'objet retourné par cette requête implémente l'interface **Query** (même package) et possède de ce fait des méthodes comme :

`java.util.List getResultList()`

qui retourne une liste contenant les résultats de la requête,

- ◆ `int executeUpdate()`

qui permet de réaliser une commande de suppression ou de mise à jour.

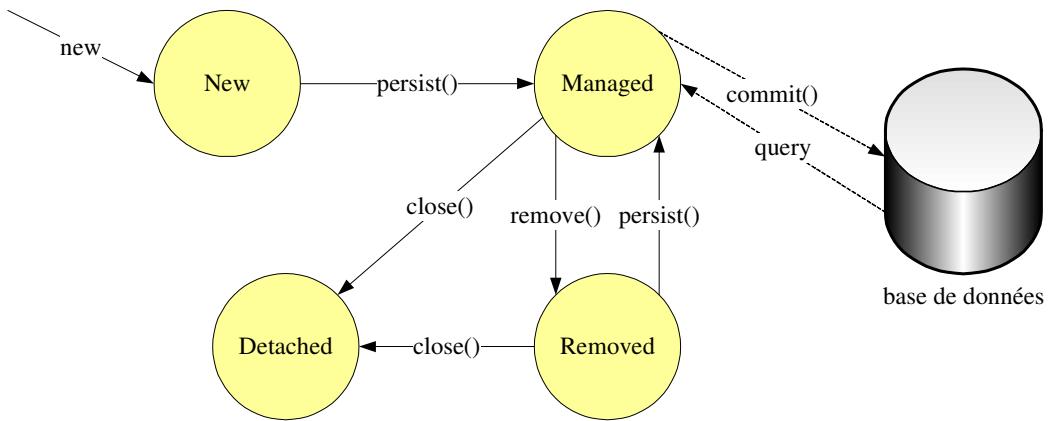
Quand on sait qu'un EntityManager possède aussi les méthodes :

`void detach(java.lang.Object entity)`
`void remove(java.lang.Object entity)`

on peut comprendre qu'un objet entity peut se trouver dans 4 états :

- ◆ **New** : il a été instancié mais n'est pas encore lié à un EntityManager;
- ◆ **Managed** : il a fait l'objet d'une commande de persistance dans une base de données (`persist()`) ou a été récupéré depuis la base de donnée;
- ◆ **Removed** : l'objet est évidemment retiré de la base;
- ◆ **Detached** : il a été déconnecté de l'EntityManager, si bien que des changements qui l'affectent ne sont plus répercutés dans la base.

Schématiquement :



L'objet implémentant EntityManager (toujours le même package) est, bien classiquement, fourni par une factory :

`EntityManager createEntityManager()`

qui est une méthode de tout objet implémentant l'interface **EntityManagerFactory**. Un tel objet factory s'obtient au moyen de la factory de la classe **Persistence** :

```
public static EntityManagerFactory createEntityManagerFactory(String persistenceUnitName)
```

la chaîne de caractères permettant d'identifier la source de données (l'"**unite de persistance**") : celle-ci peut-être le nom d'une table d'une base orientée objets (comme ObjectDB) ou être définie dans un fichier de configuration nommé persistence.xml qui contient toutes les informations nécessaires (nom d'utilisateur, mot de passe, driver JDBC, etc) - nous y reviendrons.

Reste un dernier acteur : un objet implémentant **EntityTransaction**, lequel peut être obtenu au moyen de la méthode

`EntityTransaction getTransaction()`

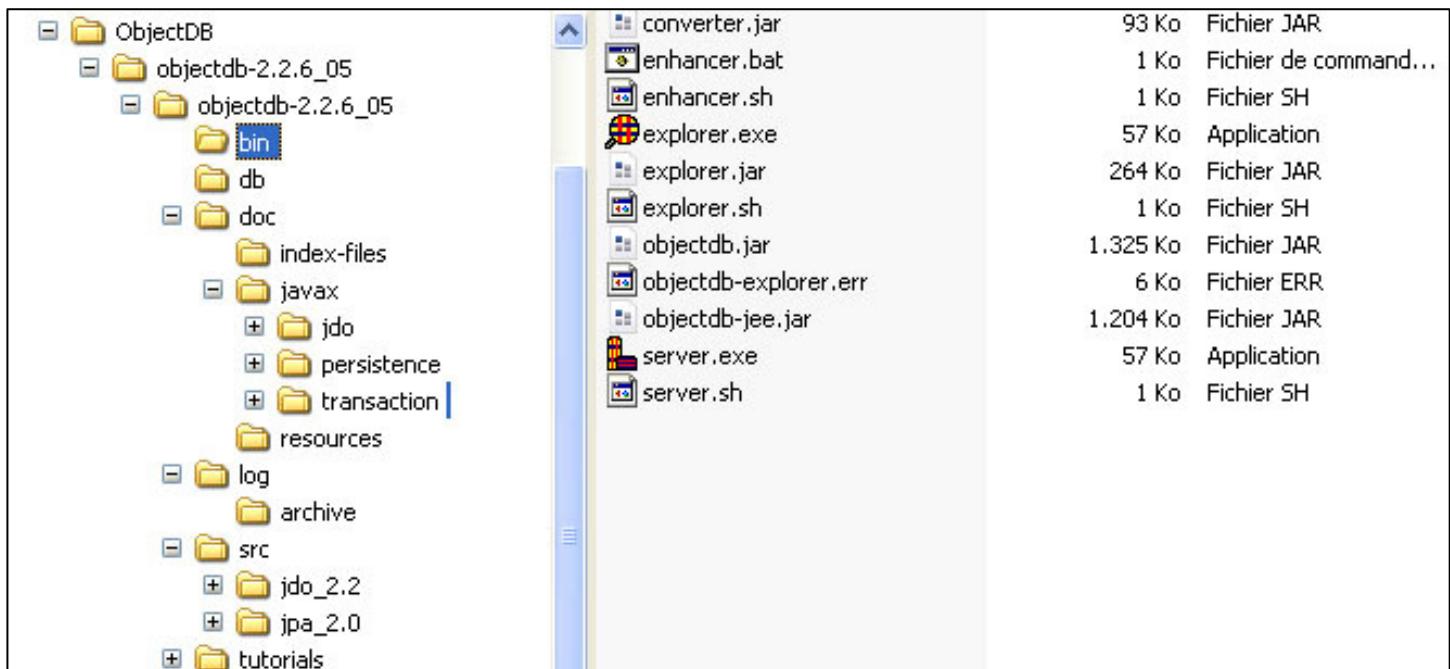
de l'EntityManager. Ave un nom pareil, on se doute de son rôle et ses méthodes le confirment :

```
void begin()
void commit()
void rollback()
```

Bien sûr, toutes ces opérations sont susceptibles de lancer des exceptions instances de toute une série de classes qui héritent de PersistenceException (toujours du même package). Comme celle-ci hérite de RuntimeException, ces exceptions sont toutes "non checkées".

3. Un exemple avec ObjectDB

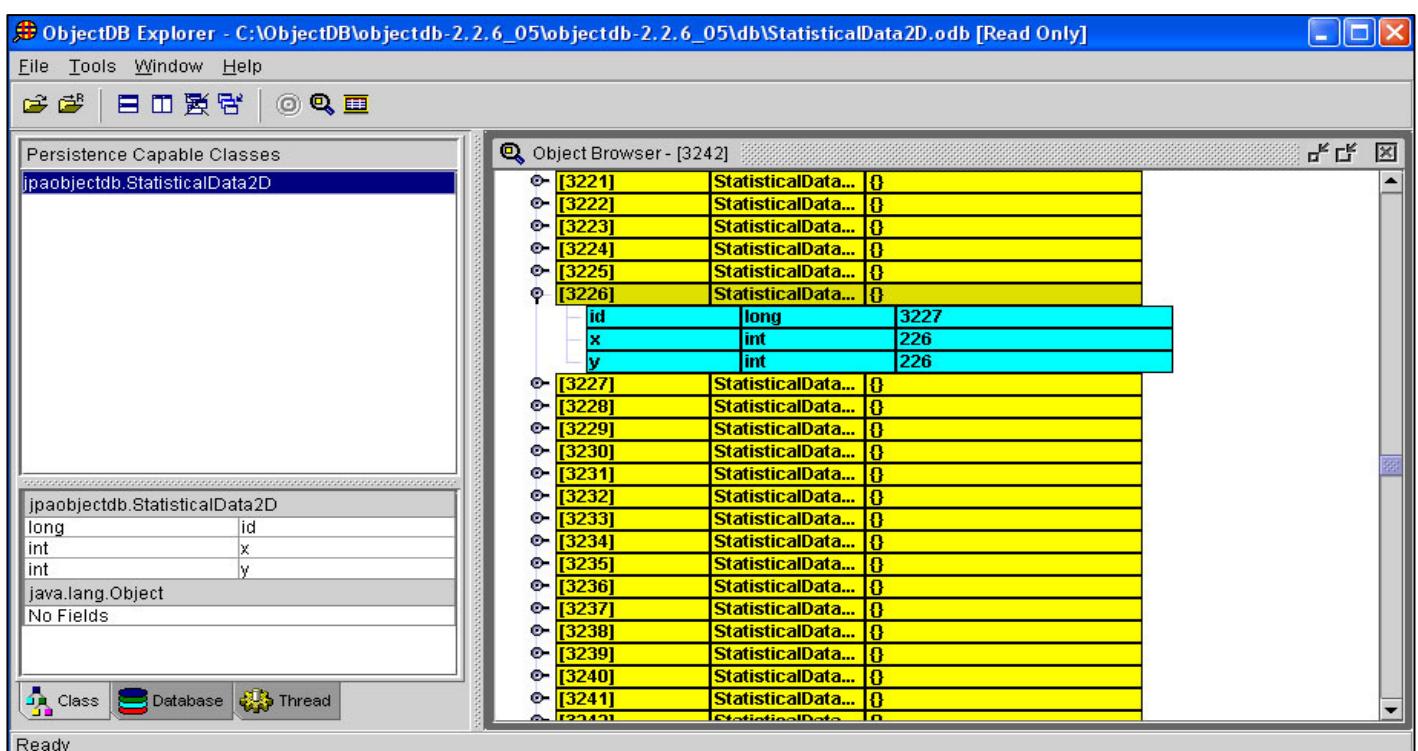
Afin de mettre de côté, provisoirement, les problèmes de configuration pour nous concentrer sur les seuls problèmes de mise en œuvre, nous allons tout d'abord utiliser un **ODBMS (Object Oriented Database Management System)** nommé ObjectDB. Outre le fait d'être pur Java, ce SGBD est géré *uniquement par programmation* au moyen des APIs JPA ou JDO. Loin d'être un jouet, ce SGBD peut, paraît-il ;-), gérer des données dont la taille atteint le TByte. Il est utilisé en production depuis 2004 et a été testé avec Tomcat, Jetty, GlassFish, JBoss et Spring. Diffusé sous forme d'un fichier objectdb-2.2.6_05.zip, il est déployable par simple décompression :



ObjectDB	converter.jar	93 Ko	Fichier JAR
objectdb-2.2.6_05	enhancer.bat	1 Ko	Fichier de command...
objectdb-2.2.6_05	enhancer.sh	1 Ko	Fichier SH
bin	explorer.exe	57 Ko	Application
db	explorer.jar	264 Ko	Fichier JAR
doc	explorer.sh	1 Ko	Fichier SH
index-files	objectdb.jar	1.325 Ko	Fichier JAR
javadoc	objectdb-explorer.err	6 Ko	Fichier ERR
jdo	objectdb-jee.jar	1.204 Ko	Fichier JAR
persistence	server.exe	57 Ko	Application
transaction	server.sh	1 Ko	Fichier SH
resources			
log			
archive			
src			
jdo_2.2			
jpa_2.0			
tutorials			

Le SGBD est fourni avec un explorateur qui permet de visualiser les objets persistants :

```
| C:\ObjectDB\objectdb-2.2.6_05\objectdb-2.2.6_05\bin>java -jar explorer.jar
```



Considérons à présent, à l'image du manuel JPA d'Apache ;-), une simple classe représentant un couple de valeurs représentant des données de statistique à deux dimensions (autrement dit, on peut les voir aussi comme des points) :

StatisticalData2D.java

```
package jpaobjectdb;

import java.io.Serializable;
import javax.persistence.*; 

@Entity
public class StatisticalData2D
//implements Serializable
{
    private static final long serialVersionUID = 1L;

    @Id @GeneratedValue
    private long id;

    private int x;
    private int y;

    public StatisticalData2D()
    {
        x=0; y=0;
    }

    StatisticalData2D(int x, int y)
    {
        this.x = x;
        this.y = y;
    }

    public long getId() { return id; }
    public int getX() { return x; }
    public int getY() { return y; }
    public void setId(long id) { this.id = id; }
    public void setX(int x) { this.x = x; }
    public void setY(int y) { this.y = y; }

    @Override
    public String toString()
    {
        return String.format("(%d, %d)", this.x, this.y);
    }
}
```

On remarquera l'annotation **@Id** qui précise quelle variable membre servira à identifier l'objet persistant (autrement dit, qui est sa clé primaire) et l'annotation **@GeneratedValue** qui

précise comment la valeur de cet identifiant doit être générée lors de la persistance : par défaut (ce que nous faisons ici), la valeur est GeneratorType.AUTO, ce qui signifie que c'est l'implémentation du vendor qui choisit le mode de génération.

On peut dès lors imaginer le programme de test suivant:

```
AnalyzeStatisticalData.java
package jpaobjectdb;

import javax.persistence.*;
import java.util.*;

public class AnalyzeStatisticalData
{
    public static void main(String[] args)
    {
        EntityManagerFactory emf =
            Persistence.createEntityManagerFactory("$objectdb/db/StatisticalData2D.odb");
        EntityManager em = emf.createEntityManager();

        em.getTransaction().begin();
        for (int i = 0; i < 10; i++)
        {
            StatisticalData2D p = new StatisticalData2D(i, 500-i);
            em.persist(p);
        }
        em.getTransaction().commit();

        Query q1 = em.createQuery("SELECT COUNT(p) FROM StatisticalData2D p");
        System.out.println("Quantité de couples : " + q1.getSingleResult());

        Query q2 = em.createQuery("SELECT AVG(p.x) FROM StatisticalData2D p");
        System.out.println("Moyenne des x : " + q2.getSingleResult());

        TypedQuery<StatisticalData2D> query =
            em.createQuery("SELECT p FROM StatisticalData2D p", StatisticalData2D.class);
        List<StatisticalData2D> results = query.getResultList();
        for (StatisticalData2D p : results)
        {
            System.out.println(p);
        }

        em.close();
        emf.close();
    }
}
```

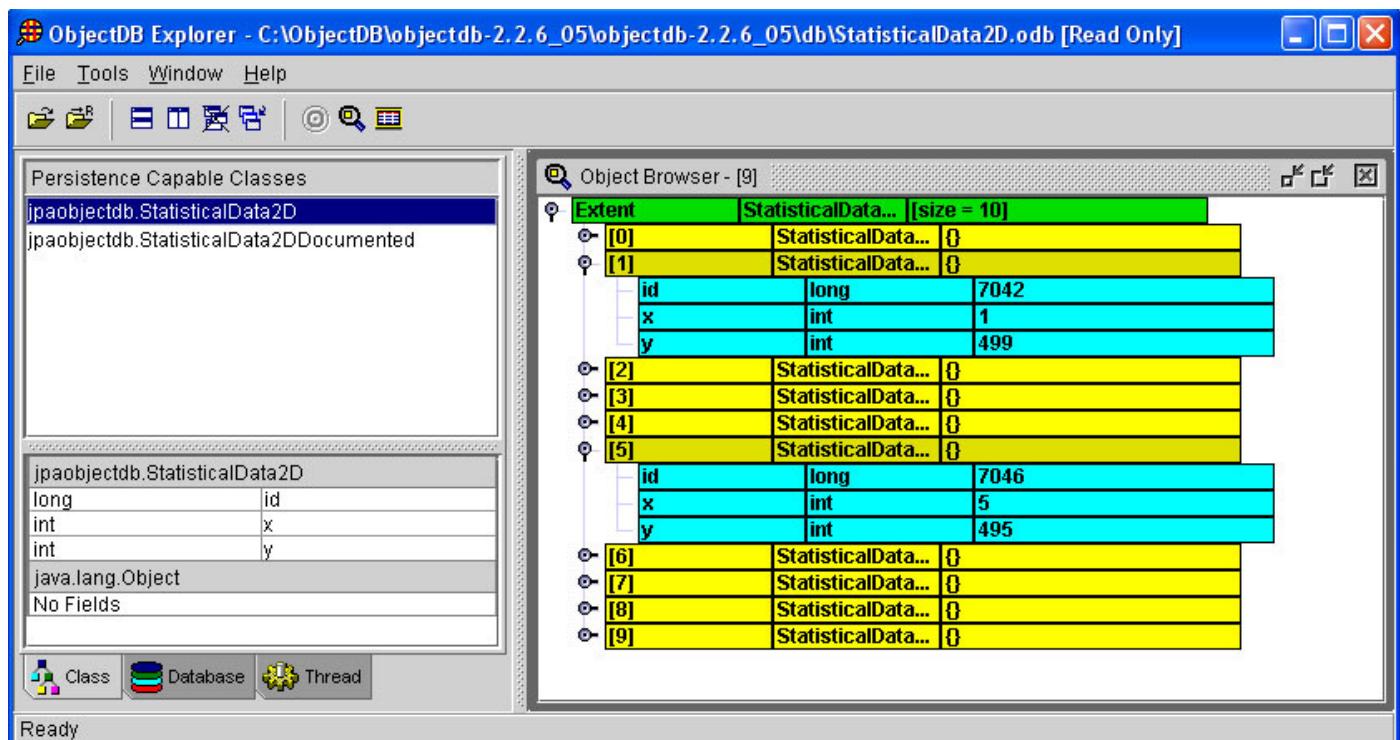
ce qui donne comme résultat à la console :

| Quantité de couples : 10

Moyenne des x : 4.5

(0, 500)
 (1, 499)
 (2, 498)
 (3, 497)
 (4, 496)
 (5, 495)
 (6, 494)
 (7, 493)
 (8, 492)
 (9, 491)

et dans l'explorer de ObjecdDB :



4. La persistance des objets agrégés

Prenons à présent comme contexte un aéroport, avec des classes prévisible **Vol** et **Avion**. Pour l'avion, aucune difficulté :

Avion.java

```
package aeroport;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
```

```

@Entity
public class Avion
{
    @Id
    private String IdAvion;
    private int nombrePassagers;

    public Avion()
    { IdAvion = "0000"; nombrePassagers=0; }

    public Avion(String id, int n)
    { IdAvion = id; nombrePassagers=n; }

    public String getIdAvion() { return IdAvion; }
    public void setIdAvion(String IdAvion) { this.IdAvion = IdAvion; }
    public int getNombrePassagers() { return nombrePassagers; }
    public void setNombrePassagers(int nombrePassagers)
        { this.nombrePassagers = nombrePassagers; }

    @Override
    public String toString()
    {
        return IdAvion + " (" + nombrePassagers + ")";
    }
}

```

Pour le vol, nous avons une variable membre (et même une propriété) *qui est elle-même une entité* : l'avion attaché au vol. Nous pouvons caractériser cette association au moyen d'annotations **@OneToOne**, **@ManyToOne** et **@OneToMany** – dans notre cas, c'est **@ManyToOne** qui convient, puisqu'un même avion peut être référencé par plusieurs vols. Ce genre d'annotation est paramétré par l'une des valeurs de l'énumération **CascadeType** :

- ◆ **CascadeType.PERSIST**: si on persiste l'objet principal, l'objet référencé l'est aussi;
- ◆ **CascadeType.REMOVE**: en cas de suppression, on supprime aussi l'objet référencé;
- ◆ **CascadeType.REFRESH**: idem vis à vis d'un rafraîchissement des données;
- ◆ **CascadeType.ALL** : cumule les valeurs ci-dessus (et aussi **CascadeType.MERGE**).

Donc :

Vol.java
package aeroport;
import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.ManyToOne;

```

@Entity
public class Vol
{
    @ManyToOne(cascade=CascadeType.ALL)
    private Avion avion;

    @Id @GeneratedValue
    long IdVol;

    public Vol()
    { avion=null; }
    public Vol(Avion a)
    { avion=a; }

    public long getIdVol() { return IdVol; }
    public void setIdVol(long IdVol) { this.IdVol = IdVol; }
    public Avion getAvion() { return avion; }
    public void setAvion(Avion avion) { this.avion = avion; }

    @Override
    public String toString()
    {
        return "[ " + IdVol + " avec l'avion " + this.getAvion().toString() + "]";
    }
}

```

On peut alors imaginer un programme rendant persistant quelques vols et leurs avions associés :

GenerateAirportData.java

```

package jpaobjectdb;

import aeroport.Avion;
import aeroport.Vol;
import javax.persistence.*;
import java.util.*;

public class GenerateAirportData
{
    public static void main(String[] args)
    {
        EntityManagerFactory emf =
            Persistence.createEntityManagerFactory("$objectdb/db/Aeroport.odb");
        EntityManager em = emf.createEntityManager();

        em.getTransaction().begin();
        Avion av1=new Avion("5214", 120); Avion av2=new Avion("5269", 250);
        Vol v1 = new Vol(av1);em.persist(v1);
        Vol v2 = new Vol(av2);em.persist(v2);
    }
}

```

```

Vol v3 = new Vol(av1);em.persist(v3);
Vol v4 = new Vol(av1);em.persist(v4);
em.getTransaction().commit();

Query q1 = em.createQuery("SELECT COUNT(a) FROM aeroport.Vol a");
System.out.println("Quantité de vols : " + q1.getSingleResult());

TypedQuery<Vol> query =
    em.createQuery("SELECT a FROM aeroport.Vol a", Vol.class);
List<Vol> results = query.getResultList();
for (Vol v : results) { System.out.println(v); }

em.close();
emf.close();
}
}

```

Sur la console :

```

Quantité de vols : 4
[ 1 avec l'avion 5214 (120)]
[ 2 avec l'avion 5269 (250)]
[ 3 avec l'avion 5214 (120)]
[ 4 avec l'avion 5214 (120)]

```

et l'explorer d'ObjectDB confirme :

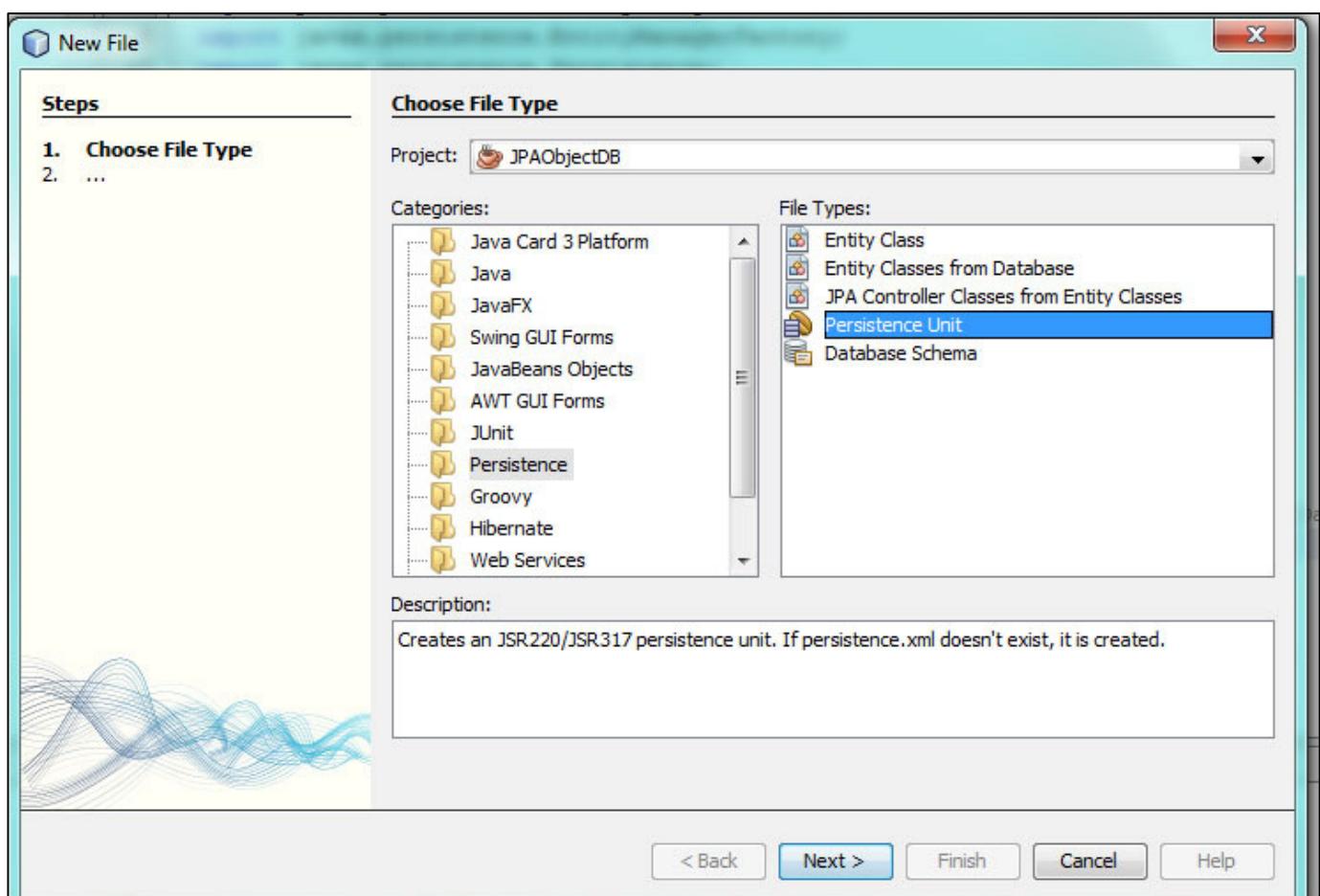
Index	Vol	IdVol	avion	nombrePassagers	idAvion
[0]	Vol#1	1	Avion#5214	0	
[1]	Vol#2	2	Avion#5269	250	"5269"
[2]	Vol#3	3	Avion#5214	120	"5214"
[3]	Vol#4	0			

5. Une unité de persistance MySQL

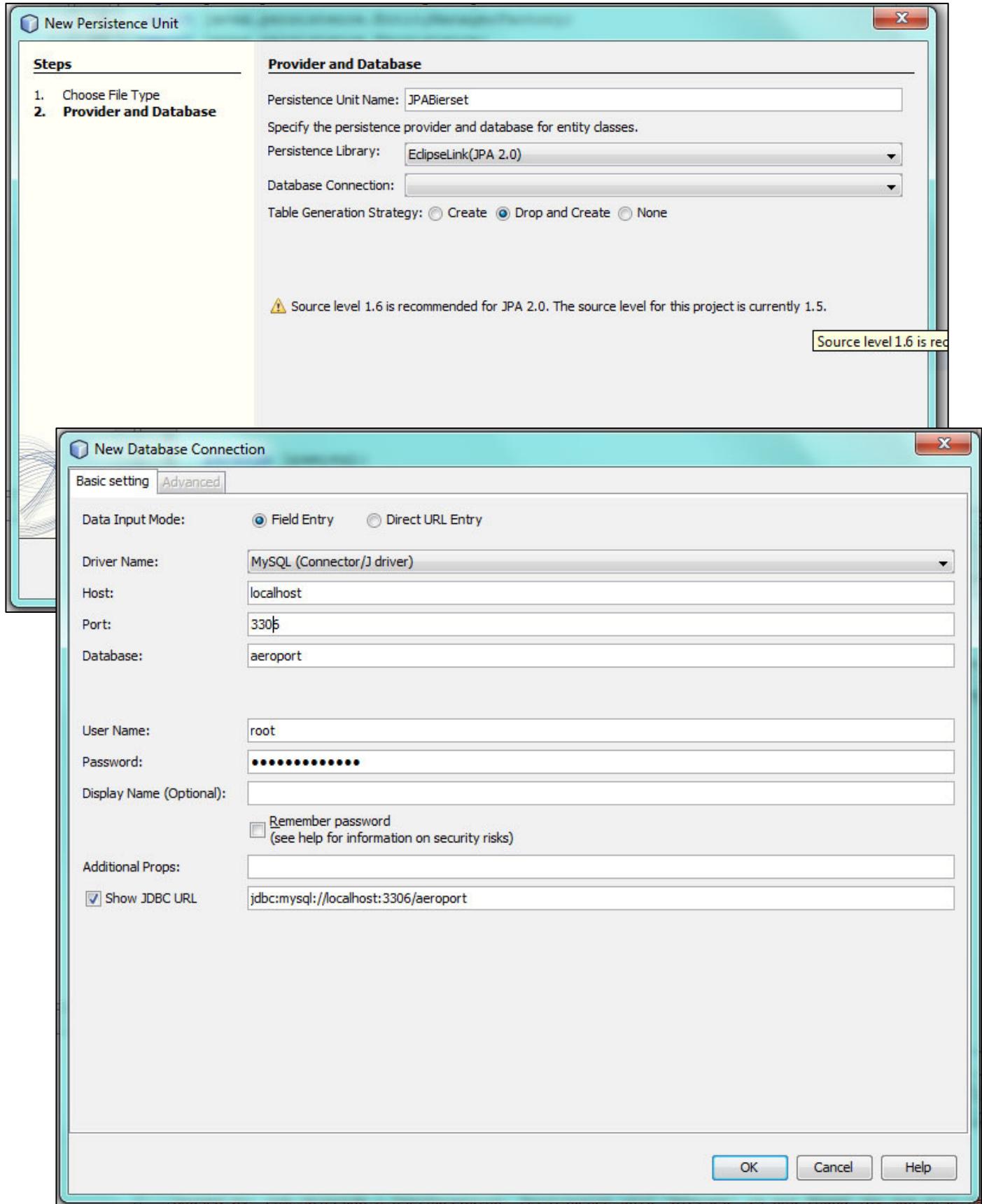
Supposons à présent vouloir créer vouloir persister nos objets dans une base de données non orientée objets, par exemple dans une base MySQL. Disons que la base modélise un aéroport et que nous souhaitons persister des objets Avion (classe ci-dessus). Il va alors falloir définir une "**unité de persistance**" [*persistence unit*] qui décrit une configuration d'accès à la base et qui, associée à l'EntityManagerFactory, permettra de créer des EntityManager qui accèderont à cette base. Cette unité de persistance est définie dans un fichier **persistence.xml** que NetBeans nous permet de créer facilement. En effet, supposons disposer d'une base de données MySQL aeroport comportant une table avion :

Field *	Type *	Collation	Null *	Key *
IDAVION	varchar(255)	latin1_swedish_ci	NO	PRI
NOMBREPASSAGERS	int(11)	{null}	YES	

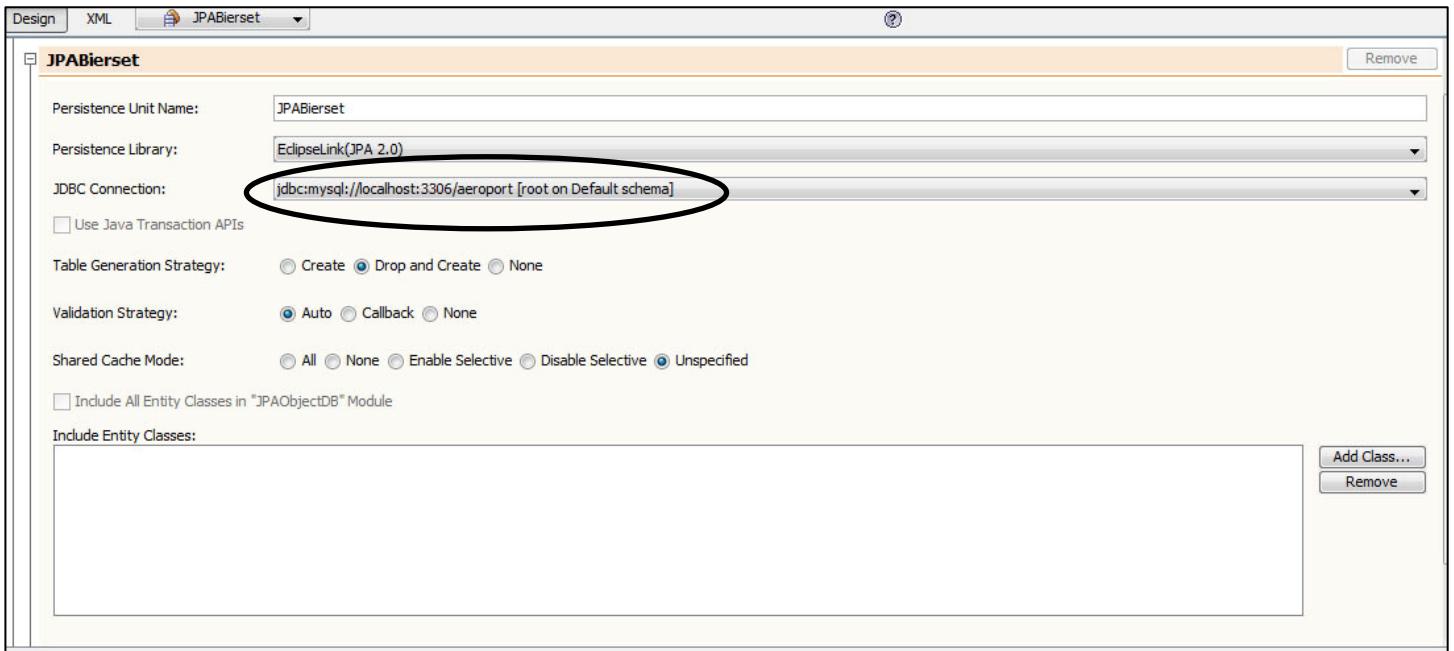
Sous NetBeans, un clic droit sur le projet :



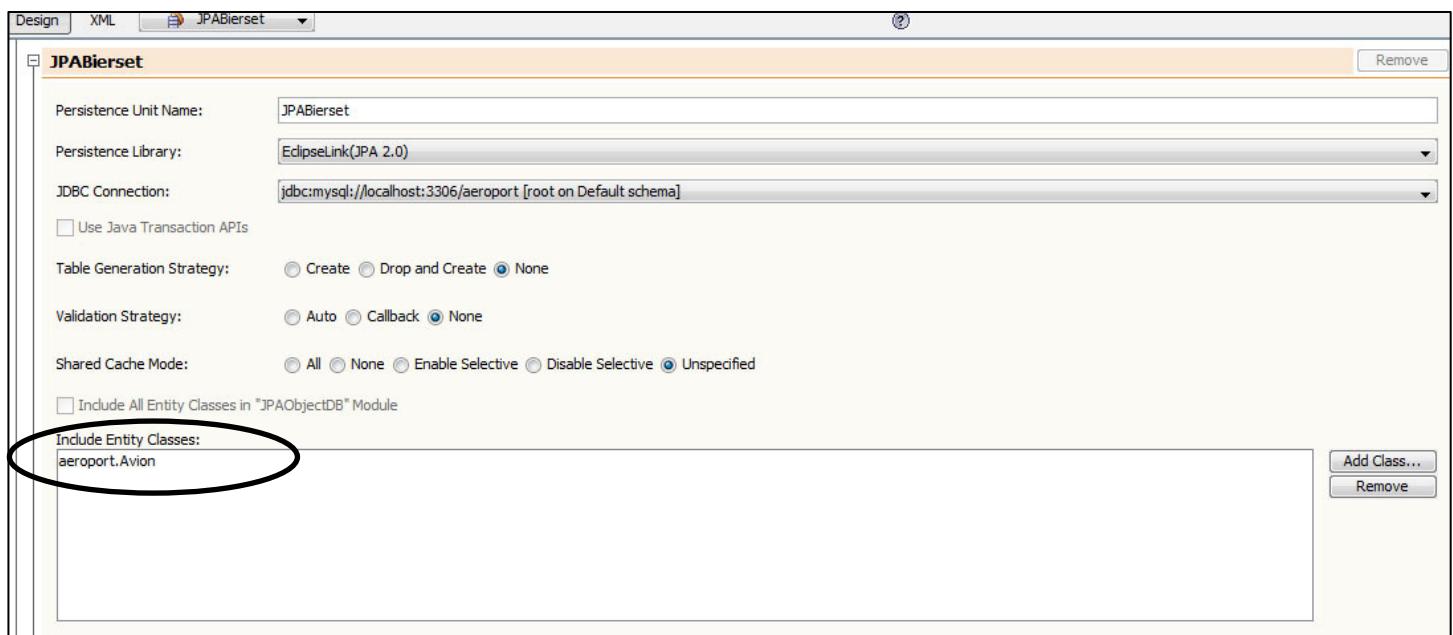
nous permet tout d'abord de choisir une implémentation de JPA (ici, celle d'Eclipse mais nous aurions pu choisir celle d'Oracle) :



On obtient ainsi un fichier XML dont la vue "design" ressemble à ceci :



On constate que l'on peut choisir la stratégie de génération de table ou de validation. Surtout, on peut (et doit) préciser quelles sont les classes Identity :



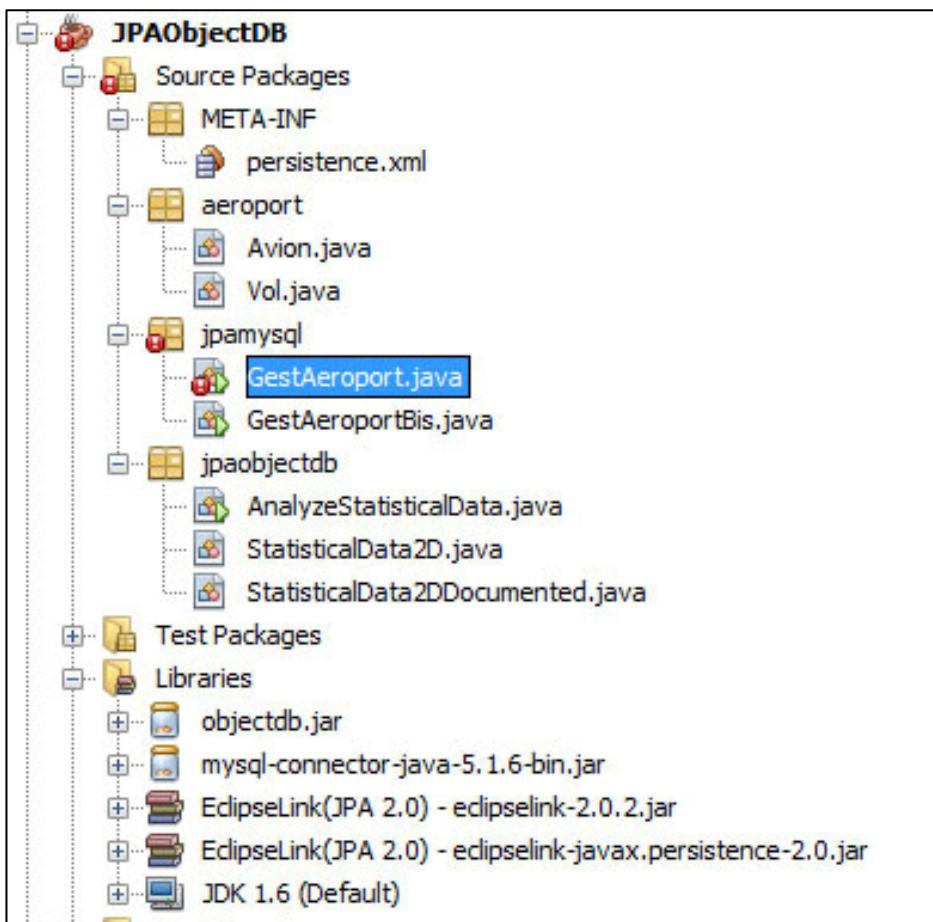
Le fichier XML (qui se trouve dans le répertoire META-INF de ...\\build\\classes) est donc en définitive :

persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0" xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
  http://java.sun.com/xml/ns/persistence_2_0.xsd">
```

```
<persistence-unit name="JPABierset" transaction-type="RESOURCE_LOCAL">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <class>aeroport.Avion</class>
    <validation-mode>NONE</validation-mode>
    <properties>
        <property name="javax.persistence.jdbc.url"
            value="jdbc:mysql://localhost:3306/aeroport"/>
        <property name="javax.persistence.jdbc.password" value="GrosZZZZZZ"/>
        <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver"/>
        <property name="javax.persistence.jdbc.user" value="root"/>
    </properties>
</persistence-unit>
</persistence>
```

Le projet ressemblera donc à ceci :



Le programme de persistance s'écrit simplement en utilisant une EntityManagerFactory qui se nourrit de notre unité de persistance :

GestAeroport.java

```
package jpamysql;
import aeroport.Avion;
```

```
import java.util.List;
import javax.persistence.Query;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import javax.persistence.TypedQuery;

public class GestAeroport
{
    public static void main(String[] args)
    {
        EntityManagerFactory emf =
            Persistence.createEntityManagerFactory("JPABierset");
        EntityManager em = emf.createEntityManager();

        em.getTransaction().begin();
        Avion av1=new Avion("5214", 120);em.persist(av1);
        Avion av2=new Avion("5269", 250);em.persist(av2);
        em.getTransaction().commit();

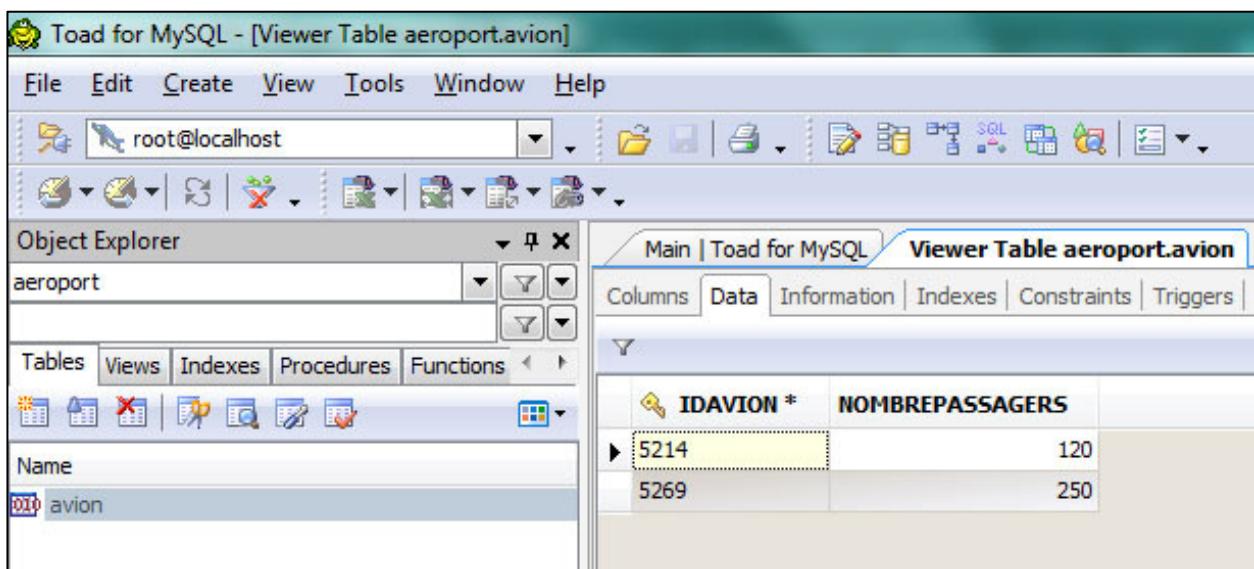
        TypedQuery<Avion> query =
                    em.createQuery("SELECT a FROM Avion a",
        List<Avion> results = (List<Avion>) query.getResultList();
        System.out.println("Nombre d'avions récupérés =" + results.size());
        for (Avion v : results)
        {
            System.out.println(v);
        }
    }
}
```

Résultat :

```
[EL Info]: 2011-06-27 12:20:49.312--ServerSession(23275591)--EclipseLink, version:
Eclipse Persistence Services - 2.0.2.v20100323-r6872
[EL Info]: 2011-06-27 12:20:49.642--ServerSession(23275591)--file:/C:/java-netbeans-
application/JPAObjectDB/src/_JPABierset login successful
```

Nombre d'avions récupérés =2
5214 (120)
5269 (250)

et on peut vérifier dans la base :



Si nous persistons à présent des objets vols, après avoir modifié l'unité de persistance pour créer des tables qui n'existent pas et connaître aussi comme Entity des objets Vol :

persistence.xml (2)

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0" xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
  http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
  <persistence-unit name="JPABierset" transaction-type="RESOURCE_LOCAL">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <class>aeroport.Avion</class>
    <class>aeroport.Vol</class>
    <validation-mode>NONE</validation-mode>
    <properties>
      <property name="javax.persistence.jdbc.url"
        value="jdbc:mysql://localhost:3306/aeroport"/>
      <property name="javax.persistence.jdbc.password" value="GrosZZZZZZ"/>
      <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver"/>
      <property name="javax.persistence.jdbc.user" value="root"/>
      <property name="eclipselink.ddl-generation" value="create-tables"/>
    </properties>
  </persistence-unit>
</persistence>
```

en utilisant le programme suivant :

GestAeroportBis.java

```
package jpamysql;

import aeroport.Avion;
import aeroport.Vol;
import java.util.List;
```

```

import javax.persistence.Query;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import javax.persistence.TypedQuery;

public class GestAeroportBis
{
    public static void main(String[] args)
    {
        EntityManagerFactory emf =
            Persistence.createEntityManagerFactory("JPABierset");
        EntityManager em = emf.createEntityManager();

        em.getTransaction().begin();
        Avion av1=new Avion("7542", 120);em.persist(av1);
        Avion av2=new Avion("7854", 250);em.persist(av2);
        Vol v1 = new Vol(av1);em.persist(v1);
        Vol v2 = new Vol(av2);em.persist(v2);
        Vol v3 = new Vol(av1);em.persist(v3);
        Vol v4 = new Vol(av1);em.persist(v4);
        em.getTransaction().commit();

        TypedQuery<Vol> query = em.createQuery("SELECT a FROM Vol a", Vol.class);
        List<Vol> results = (List<Vol>) query.getResultList();
        System.out.println("Nombre d'avions récupérés =" + results.size());
        for (Vol v : results)
        {
            System.out.println(v);
        }
    }
}

```

on obtient :

```

[EL Info]: 2011-06-27 12:20:49.312--ServerSession(23275591)--EclipseLink, version:
Eclipse Persistence Services - 2.0.2.v20100323-r6872
[EL Info]: 2011-06-27 12:20:49.642--ServerSession(23275591)--file:/C:/java-netbeans-
application/JPAObjectDB/src/_JPABierset login successful
[EL Warning]: 2011-06-27 12:20:49.693--ServerSession(23275591)--Exception
[EclipseLink-4002] (Eclipse Persistence Services - 2.0.2.v20100323-r6872):
org.eclipse.persistence.exceptions.DatabaseException
Internal Exception: com.mysql.jdbc.exceptions.jdbc4.MySQLSyntaxErrorException: Table 'avion' already exists
Error Code: 1050
Call: CREATE TABLE AVION (IDAVION VARCHAR(255) NOT NULL,
NOMBREPASSAGERS INTEGER, PRIMARY KEY (IDAVION))
Query: DataModifyQuery(sql="CREATE TABLE AVION (IDAVION VARCHAR(255)
NOT NULL, NOMBREPASSAGERS INTEGER, PRIMARY KEY (IDAVION))")

```

Nombre d'avions récupérés =4
 [1 avec l'avion 7542 (120)]
 [3 avec l'avion 7542 (120)]
 [4 avec l'avion 7542 (120)]
 [2 avec l'avion 7854 (250)]

Dans la base MySQL, on peut constater que cette exception est sans conséquence fâcheuses :

The figure consists of three vertically stacked screenshots from the Toad for MySQL interface.

- Screenshot 1:** Shows the Object Explorer for the 'aeroport' database. Under 'Tables', it lists 'avion', 'sequence', and 'vol'. The 'Data' tab of the viewer for 'aeroport.vol' shows the following data:

IDVOL *	AVION_IDAVION
1	7542
3	7542
4	7542
2	7854

- Screenshot 2:** Shows the Object Explorer for the 'aeroport' database. Under 'Tables', it lists 'avion', 'sequence', and 'vol'. The 'Data' tab of the viewer for 'aeroport.avion' shows the following data:

IDAVION *	NOMBREPASSAGERS
5214	120
5269	250
7542	120
7854	250

- Screenshot 3:** Shows the Object Explorer for the 'aeroport' database. Under 'Tables', it lists 'avion', 'sequence', and 'vol'. The 'Data' tab of the viewer for 'aeroport.sequence' shows the following data:

SEQ_NAME *	SEQ_COUNT
SEQ_GEN	50

Quel est donc l'usage de cette dernière table ? Fournir le nombre de départ pour la génération des identifiants. Ainsi, si on modifie la classe Avion comme ceci :

Avion.java (2)

```
...
public class Avion
{
```

```
@Id @GeneratedValue  
private String IdAvion;  
private int nombrePassagers;  
  
public Avion()  
{ nombrePassagers=0; }  
  
public Avion(int n)  
{ nombrePassagers=n; }  
...
```

et le programme de test :

GestAeroportBis.java (2)

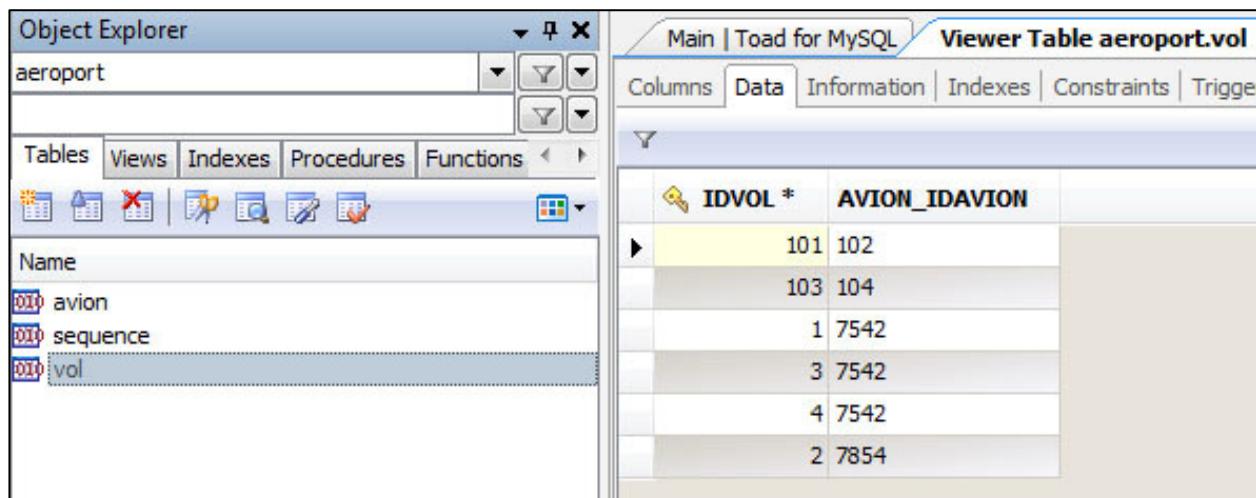
```
...  
public class GestAeroportBis  
{  
    public static void main(String[] args)  
    {  
        ...  
        em.getTransaction().begin();  
        Avion av1=new Avion(120); Avion av2=new Avion(250);  
        Vol v10 = new Vol(av1);em.persist(v10);  
        Vol v11 = new Vol(av2);em.persist(v11);  
        em.getTransaction().commit();  
        ...  
    }  
}
```

on obtient, si la table de sequence contenait 100 :

```
[EL Info]: 2011-06-27 13:14:11.87--ServerSession(16022517)--EclipseLink, version: Eclipse  
Persistence Services - 2.0.2.v20100323-r6872  
[EL Info]: 2011-06-27 13:14:12.218--ServerSession(16022517)--file:/C:/java-netbeans-  
application/JPAObjectDB/src/_JPABierset login successful  
...  
Internal Exception: com.mysql.jdbc.exceptions.jdbc4.MySQLSyntaxErrorException: Table  
'vol' already exists  
Error Code: 1050  
...  
Internal Exception: com.mysql.jdbc.exceptions.jdbc4.MySQLSyntaxErrorException: Table  
'avion' already exists  
Error Code: 1050  
...  
[EL Warning]: 2011-06-27 13:14:12.273--ServerSession(16022517)--Exception ...  
Internal Exception: com.mysql.jdbc.exceptions.jdbc4.MySQLSyntaxErrorException: Table  
'sequence' already exists  
Error Code: 1050  
...
```

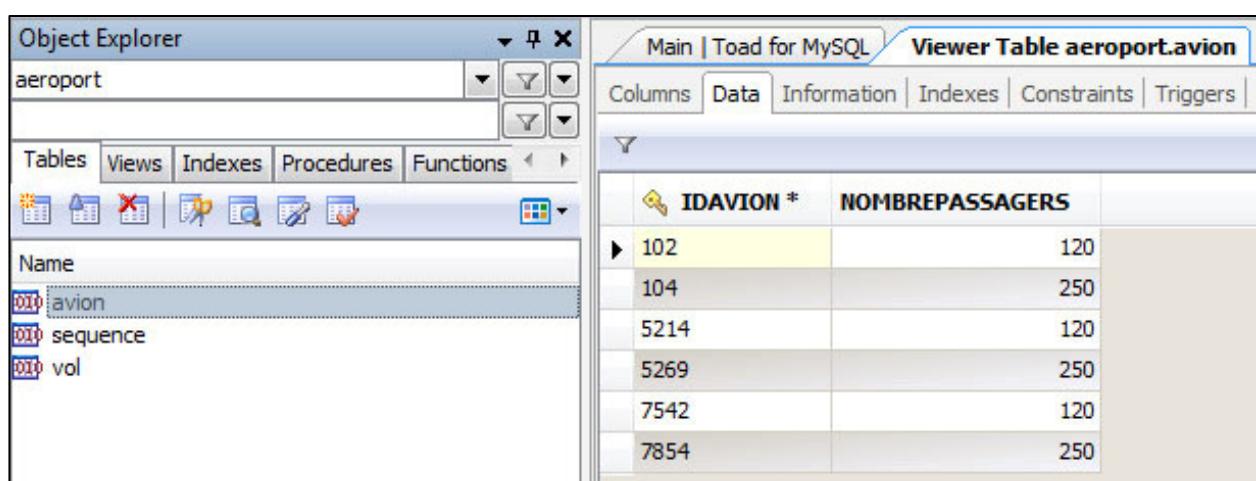
Nombre d'avions récupérés =6
 [101 avec l'avion 102 (120)]
 [103 avec l'avion 104 (250)]
 [1 avec l'avion 7542 (120)]
 [3 avec l'avion 7542 (120)]
 [4 avec l'avion 7542 (120)]
 [2 avec l'avion 7854 (250)]

Au niveau de la base, tout est correct ☺ :



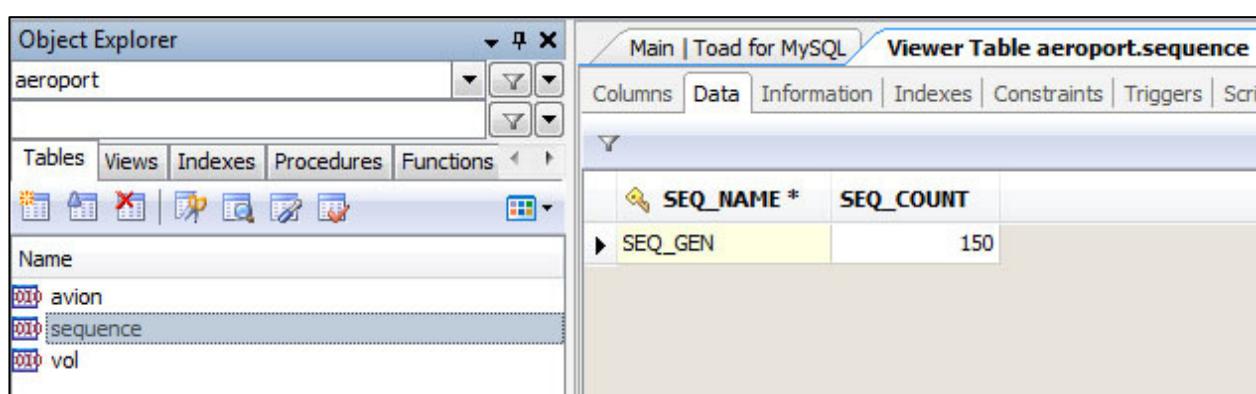
The screenshot shows the Object Explorer on the left with the 'aeroport' database selected. The 'Tables' tab is active, showing three tables: 'avion', 'sequence', and 'vol'. The 'Data' tab for the 'aeroport.vol' table is selected on the right, displaying the following data:

IDVOL *	AVION_IDAVION
101	102
103	104
1	7542
3	7542
4	7542
2	7854



The screenshot shows the Object Explorer on the left with the 'aeroport' database selected. The 'Tables' tab is active, showing three tables: 'avion', 'sequence', and 'vol'. The 'Data' tab for the 'aeroport.avion' table is selected on the right, displaying the following data:

IDAVION *	NOMBREPASSAGERS
102	120
104	250
5214	120
5269	250
7542	120
7854	250



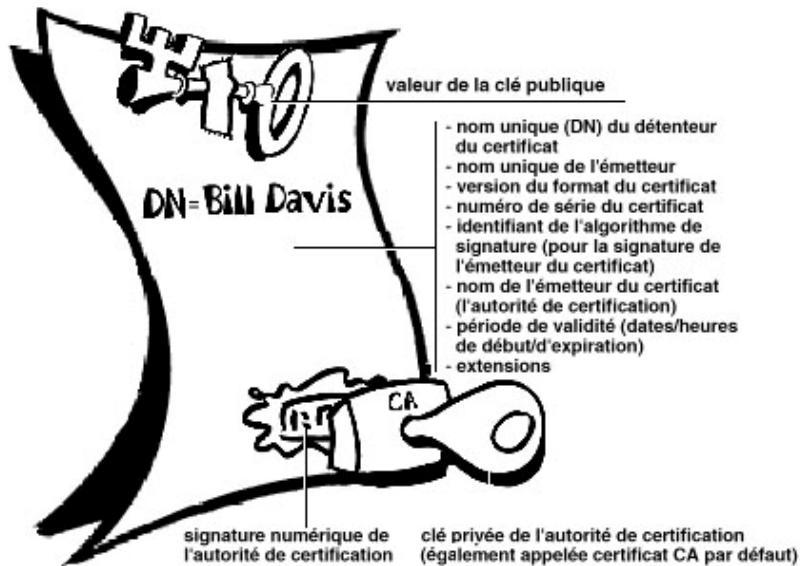
The screenshot shows the Object Explorer on the left with the 'aeroport' database selected. The 'Tables' tab is active, showing three tables: 'avion', 'sequence', and 'vol'. The 'Data' tab for the 'aeroport.sequence' table is selected on the right, displaying the following data:

SEQ_NAME *	SEQ_COUNT
SEQ_GEN	150

Annexe 4 : Les classes de certificats

Il est courant de catégoriser les certificats par leur classe. Cette classe permet de connaître quel niveau de validation (et donc de sécurité) on peut attendre de ces certificats.

Verisign définit 5 classes de certificats :



Classe 1: validation très limitée

POUR personnes individuelles pour e-mail

- ◆ Certificat serveur: seul le **whois** du domaine a été consulté,
- ◆ Certificat utilisateur/email: il n'y a que l'**adresse email** du titulaire qui est vérifiée.

Classe 2: l'organisation a été vérifiée.

L'autorité de certification garantit que l'**organisation existe**, que le nom de domaine associé lui appartient (ou qu'elle dispose des droits d'utilisation) et qu'un responsable de l'organisation ait autorisé l'émission du certificat.

POUR organisations qui doivent s'identifier

- ◆ Certificat serveur;
- ◆ Certificat développeur;
- ◆ Certificat utilisateur/email.

Classe 3: l'organisation a été vérifiée et le titulaire (la personne physique) a été authentifié **en face à face**.

Classe 3+ ("Buil in token"): classe 3 + délivrance de la clé privée et du certificat associé sur support physique: token USB ou carte à puce.

POUR serveurs et softwares à identifier au moyen d'un CA = **trusted third party**.

Classe 4: comme classe 3, mais réservé aux transactions financières entre entreprises (**B2B**).

Classe 5: comme classe 3, mais réservé aux **organisations gouvernementales**.

Ouvrages consultés

Ouvrages imprimés

Arnold, K. & Gosling, J. The Java Programming Language - Second Edition / The Java Series. Reading, Massachussettes, U.S.A. Addison-Wesley Publishing Company. 1997.

Benitez, E. Etude de nouvelles technologies intégrées dans Java Software Development Kit 1.4 (JDBC & JSSE). Seraing, Belgique. TFE In.Pr.E.S. 2004.

Campione, M. & Walrath, K. The Java Tutorial - Object-oriented Programming for the Internet / The Java Series. Reading, Massachussettes, U.S.A. Addison-Wesley Publishing Company. 1997.

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.

Eckstein, R, Loy, M. & Wood, D. Java Swing. Sebastopol, California, U.S.A. O'Reilly and Associates, Inc. 1998.

Englander, R. Developping Java beans. Sebastopol, California, U.S.A. O'Reilly and Associates, Inc. 1997.

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.

Fusco, B. Etude de la migration en Java d'une librairie C++ de IRC Group. Seraing, Belgique. TFE In.Pr.E.S. 2005.

Harold, E. R. Programmation réseau avec Java. Sebastopol, California, U.S.A. O'Reilly and Associates, Inc. 1997.

Herbiet, L. Etude de l'API de sécurité du langage Java et de son extension cryptographique. Seraing, Belgique. TFE In.Pr.E.S. 2000.

Jeunesse, F. Etude synthétique de techniques de sécurité utilisées par les applications réseaux. Seraing, Belgique. TFE Haute Ecole de la Province de Liège (In.Pr.E.S.). 2009.

Knudsen, J. Java cryptography. Sebastopol, California, U.S.A. O'Reilly and Associates, Inc. 1998.

Konat, S. Gestion de la réservation en ligne des places d'une salle de spectacle. Seraing, Belgique. TFE In.Pr.E.S. 2002.

Leburton, C. Utilisation des technologies Java et XML dans le cadre d'un projet européen : Onto-Logging. Seraing, Belgique. TFE In.Pr.E.S. 2002.

Lindholm, T. & Yellin, F. The Java Virtual Machine Specification Second Edition. The Java Series. Reading, Massachusetts, U.S.A. Addison-Wesley Publishing Company. 1999.

Machiroux, J-C. Etude des APIs d'extension Java : La programmation des GUIs selon les JFC et leur comparaison avec X Window – Application aux extensions XML. TFE In.Pr.E.S. 2002.

Schneier, B. Applied cryptography. New-York, U.S.A. John Wiley & Sons, Inc. 1994.

Schneier, B. Secrets and lies. New-York, U.S.A. John Wiley & Sons, Inc. 2000.

Singh, S. The code book – The science of secrecy from Ancient Egypt to quantum cryptography. London, United Kingdom. Fourth Estate Ltd. 1999.

Vilvens, C. Langage Java (I) : Programmation de base. Seraing, Belgique. 2020.

Vilvens, C. Programmation TCP/IP en C/C++, Java et C#/.NET. Seraing, Belgique. 2020.

Vilvens, C. Le protocole HTTP et le langage HTML. Seraing, Belgique. 2020.

Walrath, K. & Campione, M.. The JFC Swing Tutorial – A guide to constructing GUIs / The Java Series. Reading, Massachusetts, U.S.A. Addison-Wesley Publishing Company. 2001.

White, S., Fischer, M., Catell, R., Hamilton, G. & Hapner, M. JDBC API Tutorial and Reference, Second Edition / The Java Series. Reading, Massachusetts, U.S.A. Addison-Wesley Publishing Company. 1999.

sans oublier les magazines :

Pour la Science. L'art du secret. Dossier n°36, juillet-octobre 2002.

Science & Vie Junior. Les codes secrets. Hors-série n°53, juillet 2003.

et aussi

Java Cryptography Architecture API Specification & Reference
from JDK 1.6

Sites Internet

<http://java.sun.com/>

➔ <http://www.oracle.com/technetwork/java/index.html>

avec en particulier :

https://community.oracle.com/community/java/java_desktop



<https://jcp.org/en/home/index>



<https://www.meetup.com/fr-FR/Belgian-Java-User-Group/>

Site du Belgian Java User Group

et

<https://openclassrooms.com/fr/courses/2654601-java-et-la-programmation-reseau>

Java et la programmation réseau

<http://www.rsa.com>

Pour une explication complète de l'algorithme RSA

<http://www.ietf.org/rfc>

Site officiel du texte des RFC

<http://www.bouncycastle.org>

Le site de Bouncy Castle, provider de classes de sécurité.

<http://www.bibmath.net/crypto/>

Un site d'introduction avec de nombreux exemples historiques

<http://fr.wikipedia.org/wiki/Portail%3ACryptologie>

Toute une série de définitions précises

<http://openjpa.apache.org/index.html>

Le site officiel avec manuel pdf

<http://www.objectdb.com/tutorial/jpa/start>

Le site de ObjectDB

<https://humbert-florent.developpez.com/java/reseau/avance/>

F.Humbert: Avec notamment les SocketChannel sur developpez.com

<https://gfx.developpez.com/tutoriel/java/network/>

R. Guy. Architecture Client/Serveur en Java avec les sockets

